Project N°: FP6-015905

Project Acronym: MOBIUS

Project Title: Mobility, Ubiquity and Security

Instrument: Integrated Project

Priority 2: Information Society Technologies

Future and Emerging Technologies

Deliverable 3.10

Final Report on Program Verification Environment and Annotation Generation

Due date of deliverable: 2009-09-30 (T0+48)

Actual submission date: 2009-09-xx

Start date of the project: 1 September 2005 Duration: 48 months

Organisation name of lead contractor for this deliverable: UCD

Revision: Creation

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to chapter
ETH	2
INRIA	2 6
RU	2 6
UCD	1 2 3 4 5 7
WU	2 3 6 7

This document has been written by Julien Charles (UCD), Dermot Cochran (UCD), Werner Dietl (ETH), Fintan Fairmichael (UCD), Radu Grigore (UCD), Marieke Huisman (INRIA), Joe Kiniry (UCD), Erik Poll (RU), and Aleksy Schubert (WU).

Executive Summary:

Final Report on Program Verification Environment and Annotation Generation

This document describes the results of Tasks 3.6 and 3.7 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. The focus of these tasks is the design and development of an integrated program verification environment (PVE), which is colloquially called the MOBIUS PVE. Full information on this project, including the contents of this deliverable, is available online at http://mobius.inria.fr/. More information on the MOBIUS PVE is available online at http://mobius.ucd.ie/.

Task 3.6 develops a program verification environment for the automatic and interactive verification of Java programs, integrating the techniques developed in WP3. Such an environment is needed to tackle the realistic case studies in WP5, and, more generally, is crucial if we are to seriously evaluate the MOBIUS PCC infrastructure.

The MOBIUS verification environment supports the development of specifications, programs, and proofs of their security properties. It supports both source code and bytecode level verification, and is able to produce the output necessary for the generation of PCC certificates.

The result of this task in the four years of the project has been five releases (called the alpha, beta, gamma, delta, and epsilon releases) of the platform, as well as dozens of independent releases of its various subsystems. These initial releases support reasoning about source code and bytecode and focus on single-threaded Java only. The PVE also generates proof obligations for a single proof generation tool. The environment also interfaces to several automated theorem provers and interactive proof assistants.

Task 3.7 focus on generating JML annotations for existing source code and bytecode. Automata or state machines are a convenient and accessible specification formalism for many properties. The techniques developed focus on generating JML annotations from a simple automaton that specifies security properties. If a program that is monitored by the automaton does not go "wrong," then the generated annotations will not be violated during any execution of the program. The algorithm and its correctness proof are formalised with a theorem prover. The generated annotations are then propagated, to assist static verification. A simple version of the propagation algorithm has been implemented.

Contents

Chapter 1

Overview of the Program Verification Environment

Many large software engineering projects have functional specifications that are thousands of pages long. These are usually written in a natural language, such as English, and with semi-formal notation such as diagrams and tables. There are often subtle ambiguities or internal contradictions within the software requirements, which are often not discovered until very late in the software development life cycle, simply because the project becomes too big for any one developer or analyst to fully understand and remember all the relevant details at the same time.

Popular high-level "modeling" languages in current use (e.g., UML) are not amenable to formal analysis because of a lack of semantics and interoperable tool support. Even so, high-level representations are compelling and useful in modern software engineering practice and software engineers regularly struggle with using these inadequate languages, even for mission critical applications.

The Mobius PVE is a customized version of the Eclipse platform that integrates a set of tools that support software development using state-of-the-art formal methods and software engineering techniques. The Mobius PVE supports the specification and verification of JML-annotated Java programs at the source code level as well as at the bytecode level using logic- and type-based techniques and the Proof-Carrying Code paradigm. The properties of particular interest include security (e.g., data is secret and never leaked) and resource guarantee properties (e.g., this method will never use more than this much memory or that much CPU) as well as total functional correctness. The Mobius PVE is thus, in a sense, a prototype realization of the U.K. Grand Challenge 6 Verifying Compiler, specialized to the domain of Java programs [HM05].

The Mobius PVE integrates theoretical and technical best-practices, leveraging the hard work of many research groups within, as well as outside of, the MOBIUS grant consortium. Many Java software engineers use the Eclipse development environment, together with plugins such as FindBugs and PMD [Ecl]. The Mobius PVE is bundled with Eclipse and these well-known third-party plugins so that Java developers are presented with a pre-configured development environment without the need to manually install each plugin.

For example, a Java developer might use the Mobius PVE to derive a JML specification from a high-level functional specification. The JML subsystem of MOBIUS might then be used to typecheck the JML specification, and, in a complementary fashion, the ESC/Java2 used look for inconsistencies in the specification. A variety of other pre-configured subsystems like CheckStyle, FindBugs, PMD, and others are executed asynchronously while the developer writes, modifies, and evolves their software architecture. Feedback about the completeness and quality of the system's documentation, formal specification, and program code are always at the developer's fingertips. Once the Java code has been verified against the JML specification, both statically using ESC/Java2 and dynamically using hand-written and automatically generated unit tests using JML's runtime assertion checking, then the developer has very high confidence that the requirements have been implemented correctly. At this stage, one performs automatic and interactive verification using the MOBIUS DirectVCGen and Prover Editor, fully verifying functional and non-functional properties.

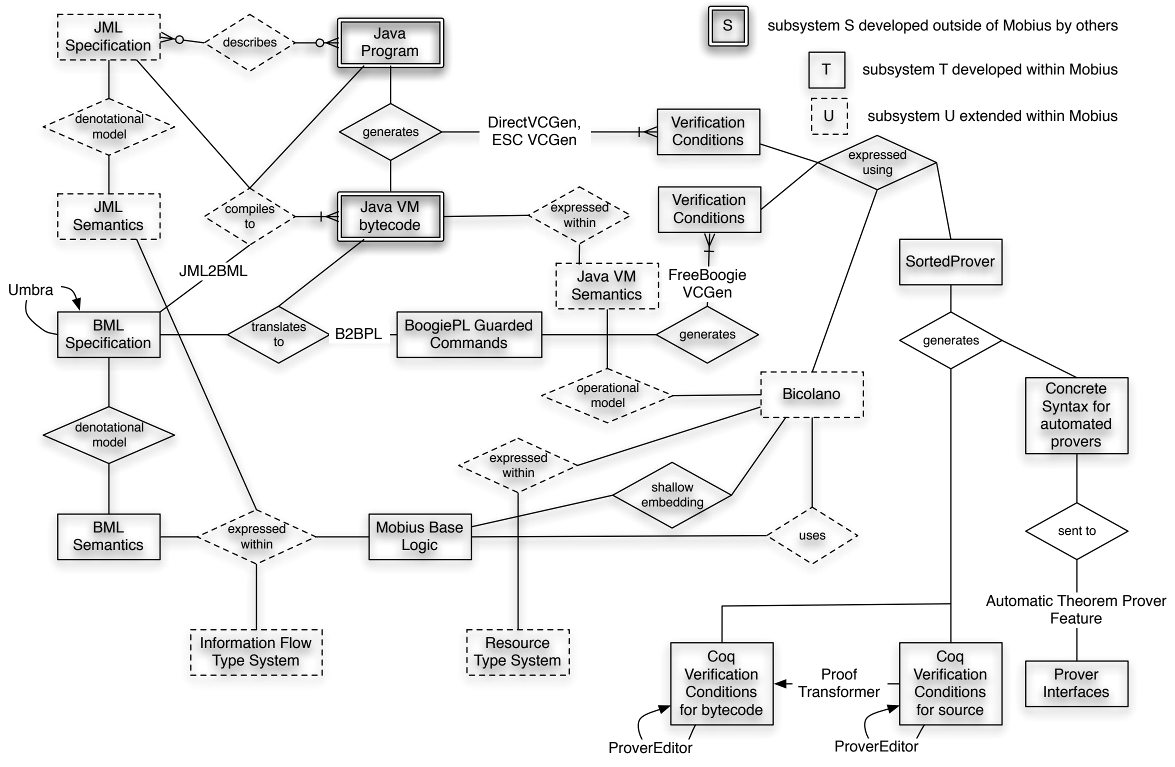


Figure 1.1: Schematic overview of the different formats (for code and proof obligations), and the translations between them supported by the PVE and in Coq. See the key for an explanation of the diagram formatting.

Recent editions of the Mobius PVE has been used at several tutorials run at international conferences including ETAPS, the GLOBAN Summer School run by MOBIUS partner UW, in courses taught at UCD, and at the JML Spec-a-thon held at the University of Washington, Tacoma in May 2009.

Chapter 2

Subsystems of the MOBIUS Program Verification Environment

The Mobius PVE is built on top of the Eclipse platform. The primary units of software within Eclipse are known as features, which are in turn composed of one or more plugins. We generally call these two kinds of units subsystems in the Mobius PVE documentation.

The following sections summarize all of these subsystems at a high level. More information on each subsystem is available via the MOBIUS Trac server (available via http://mobius.ucd.ie/), within publications related to specific tools cited appropriately, and from the original development team, in the case of externally developed tools.

The Mobius PVE is a customized version of Eclipse that contains over forty subsystems dedicated to reliable, dependable software development. Over half of these subsystems were implemented by MOBIUS partners.

Development on some of these subsystems (e.g., the static checker ESC/Java2) began before the project started, but their development continued within the project as their capabilities directly complemented the goals of the Mobius PVE. Other subsystems (e.g., the Umbra bytecode subsystem) were developed entirely within and for the MOBIUS project and the Mobius PVE.

This chapter contains a brief summary of each subsystem in the Mobius PVE. There are two forms of subsystems: those (co-)developed by MOBIUS partners (known as "MOBIUS subsystems"), and those developed outside of MOBIUS (known as "non-MOBIUS subsystems"); section 2.1 discusses the former, and section 2.2 the latter. Subsystems developed within MOBIUS are often discussed in more detail unless a non-MOBIUS subsystem has some special interaction with a MOBIUS one.

Some subsystems are exposed to the verification application developer via a programmatic API. Others are full-fledged Eclipse features and/or plugins. Many subsystems, even those that are plugins, also have rich command-line interfaces and are executed as normal processes.

In general, all MOBIUS subsystems written in Java and the majority of them were developed with various version of the Mobius PVE itself. Moreover, some of them have full formal specifications and have been partially or fully validated (e.g., extensively unit tested) and/or verified (e.g.,, ESC/Java2 reports no errors) with the Mobius PVE. When a subsystem has one of these properties, it is highlighted in the text.

As discussed in section 4.2, the source code, documentation, version control logs, and bug and feature tracking database of all subsystems is available online via our Trac-based

CDE located at http://mobius.ucd.ie/. All subsystems developed within MOBIUS are Open Source and released under one of several license types: the GPLv2, a BSD-style license, or the Eclipse Public License.

2.1 Subsystems Developed within MOBIUS

This section summarizes in some detail all of the subsystems that ship as part of the current "epsilon" release of the Mobius PVE. Most of these subsystems are still under development and are evolving to match the needs of Mobius PVE users and MOBIUS partners and their research and industry collaborators.

Information about these components (their description, status, location in the source repository, etc.) is also summarized in the MOBIUS Trac wiki via the Components wiki page.

Many MOBIUS subsystems are developed by MOBIUS partners working in collaboration with non-MOBIUS researchers throughout North America and Europe.

Many subsystems have interdependencies, many of which grew organically over several years of development, as one plugin will depend upon the availability of another. In fact, teasing out, documenting, and formalizing these dependencies in plugin descriptors has been a main organizational effort in Mobius PVE development.

Subsystems are of in several "flavors":

- First, there are the Java Modeling Language (JML) subsystems, containing all the tools used to type check the annotations, ease the use of JML from within Eclipse, and support runtime assertion checking.
- Second, there are the static analysis subsystems, namely the extended static checkers (like ESC/Java2 or FreeBoogie) and some annotations generators, like Houdini.
- Third, we define the automated theorem prover subsytems, which is a packaging of different theorem provers, as well as an Application Programming Interface for some of them.
- There is also the ProverEditor subsystem which gives a graphical user interface for the prover Coq.

As this set of subsystems are related to static verification, they can be easily used outside the MOBIUS project and, for the most part, are actually used. There are two other subsystems that were developed expressly for the PCC framework.

- The Bytecode Modelling Language (BML) subsystems, which offer facilities to annotate Java-bytecode with similar annotations as the Java Modeling Language, and manipulate these annotations.
- The DirectVCGen subsystems, which are a series of subsystems that implement a verification condition generator over source code and bytecode. This subsystem is designed such that one is able to reuse a proof done over the source code to solve the verification condition generated from its bytecode.

Some subsystems lie along multiple facets of these dimensions. For example, many subsytems are interactive, but also have APIs. Others, tie together every facet. For example, in the case of ESC/Java2, it reasons about Java source code, bytecode, and JML, contains several different ASTs, provides several APIs, uses many backends, and finally, is also an interactive plugin.

2.1.1 Java Modeling Language Subsystems

Several different subsystems are included that lex, parse, represent, process, and reason about the Java Modeling Language (JML), as JML is core to program specification and verification within MOBIUS [LPC$^+$05]. The inclusion of several subsystems with overlapping functionality is, in nearly all instances, more a historical artifact than a conscious developer choice.

The OpenJML Plugin

OpenJML is a suite of JML tools that supports modern Java . It provides a typechecker for JML annotations as well as a runtime assertion checker. It includes also jmldoc, an extension of javadoc that uses the JML annotations to make the documentation more accurate. It is built on top of OpenJDK, an open source implementation of javac from Sun Microsystems. It has a graphical user interface in the form of an Eclipse plugin.

The tool is still in development, but soon will see a proper first version released. Hence, the runtime assertion checking is not totally completed and the extended static checking is rudimentary implemented. Typechecking is provided for most of the JML constructs except for model programs, constraint clauses, and accessible clauses.

OpenJML should shortly have a fully operational typechecker and runtime assertion checker, consequently it will replace JML4 and JML2, which ware included in the previous versions of the Mobius PVE. JML2 is being replaced because it only support Java versions 1-1 to 1.4, and not Java 1.5 and beyond.

The JML2 Eclipse Plugin

This plugin provides a simple integration of the common JML tools into Eclipse 3.3. It is the main interactive plugin used by MOBIUS developers within the Mobius PVE at this point in time.

The following features are provided:

- simple installation of the core JML tools, including
- executing the JML checker,
- executing the JML runtime assertion checker (RAC),
- executing a program that was compiled with RAC, and
- configuration dialogs for JML checker and RAC.

The current version of the plugin is available from an ETHZ website and will be integrated into the main MOBIUS releases website and Trac in the coming year.

The JML Folding Editor

Folding editors allow one to manually or automatically show or hide various structured pieces of a document—for example, hiding the contents of a section while only keeping the section title, which is similar to hiding the implementation of a method while keeping its signature. The process of hiding and showing a usually larger or complex construct behind a concise visual representation is called "folding".

Basic folding editors permit one to manually fold and unfold constructs like source code documentation and implementation structures like method and loop bodies. Advanced folding editors provide additional features like saving and restoring the state of folding when a file is closed and then opened again, or automatically folding a set of related constructs. An example of the latter is the "short" or "contract" form supported by IDEs like EiffelStudio, whereby the implementations of all features (fields and methods) of a class are folded and hidden from the developer—only natural language documentation and contracts remain in view.

The JML Folding Editor, known under the development code name as "Grok," is an Eclipse plugin that has built-in support for folding JML specifications in an intelligent fashion. It is based upon the Open Source Coffee-Bytes folding plugin.

Manual manipulation of folding is supported, as is customizable defaults in folding behavior. For example, when a file is initially opened, should loop bodies be folded or unfolded? Perhaps only all private method bodies should be folded?

The MOBIUS extensions to this plugin focus on support for folding, in an intelligent fashion, JML formal specifications. For example, groups of adjacent related specifications written in line comments are folded as a unit. Furthermore, the state of folding within a file is saved when the Mylyn subsystem subsection 2.2.1 is in use. Finally, some basic notions of fisheye views [Fur86] have been included in the subsystem.

Our next steps in the research work focus on further evolving and experimenting with fisheye notions of Degree of Interest in the context of software systems with formal specifications. Given the numerous syntactic and semantic relationships that exist between code constructs and specification constructs, and observing the ways in which verification-centric software development and verification takes place, there are many fertile opportunities for good computer-human interaction research.

2.1.2 Static Analysis Subsystems for Extended Static Checking

ESC/Java2 has been under development for over a decade. The original ESC/Java was developed by a team at Digital Equipment Corporation's Systems Research Center in the mid-1990s. After that team disbanded and DEC-SRC was shut down (after Compaq bought Digital and then Hewlett-Packard bought Compaq), Hewlett-Packard released the source code of ESC/Java for research and educational purposes.

At that time, a team at Radboud University Nijmegen, led by Joe Kiniry, restarted research and development on Extended Static Checking for Java, and have since then released over fifteen versions of ESC/Java2. ESC/Java2 shares only a few subsystems with the original ESC/Java.

This section summarizes the core subsystems of the current release of ESC/Java2 (2.0b4) included with the Mobius PVE that remain from the original ESC/Java. All other extensions to ESC/Java2 that have taken place over the past five years are described in other sections, as they have all been incorporated into the Mobius PVE itself. These extensions include the new VC ASTs, the Automated Prover API, the new Fx7 and Z3 backends, universes typechecking, and support for higher-order extended static checking, which is not discussed in this deliverable.

ESCJava2

The Extended Static Checker for Java version 2 (ESC/Java2) is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations. Users control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments called pragmas.

ESC/Java2 is a large-scale subsystem within the Mobius PVE. Its architecture relies upon several of the other subsystems described in this document including the original DEC-SRC Java Frontend and the Simplify theorem prover, both of which are described in this section, and several other new MOBIUS subsystems, most of which are described in later sections of this chapter.

In a nutshell, ESC/Java2 translates JML-annotated Java source code into verification conditions that are passed to an automated theorem prover. Responses from the prover are translated back into informational messages useful to a normal programmer.

To implement these features, ESC/Java2 uses a combination of Javafe to lex and parse Java sourcecode, an internal bytecode subsystem to parse Java bytecode, and its own JML lexer and parser, all of which produce a JML-annotated Java AST. This representation is then translated into a semantically equivalent guarded command, dynamic single assignment-based representation, which is then used to generate a verification condition with either a weakest precondition or a strongest postcondition VC generator. This VC is represented in one of two forms, the original DEC-SRC VC AST, or the new MOBIUS sorted VC AST. This VC is passed to an automated theorem prover via one of two mechanisms, either the original DEC-SRC Simplify-specific interface, or the new MOBIUS automated prover backend. Likewise, responses from the automated prover flow back through these same interfaces.

In summary, ESC/Java2 uses more MOBIUS subsystems than any other component, except the Mobius PVE itself. It represents the core system used to perform research on, and evolve existing, generic verification functionality, as well as exercise and extend various other Mobius PVE subsystems.

ESC/Java2 is available in two forms:

- ESC/Java2 as a command-line tool with a simple Swing GUI front-end, or
- ESC/Java2 as an Eclipse 3.3 (Europa) plugin.

This latter, plugin-based version of this subsystem has several features that deeply integrate it into the MOBIUS PVE.

With respect to environment configuration, its preferences system is integrated into the Mobius PVE at the IDE-level and at the project-level. The former supports configuration at the systems-level, the latter settings specific to a project that are saved with the project.

Focusing on use, the integration with development is transparent. Each time a file is saved the extended static checker is run in the background asynchronously. Thus, from the developer point of view, warnings and errors appear as regular Eclipse markers on lines of code and as red underlines within program code and specifications. Likewise, all developer feedback appears within the appropriate Eclipse "Warning" and "Error" panes. The extended static checker is also executed manually on a variety of different constructs (on a package, a class, a method, etc.) with a single mouse action.

Javafe: The Original DEC-SRC Java Frontend

Javafe is the original DEC-SRC Java Frontend. It lexes and parses Java 1.5 bytecode and Java 1.4 source code. It is used by several other MOBIUS subsystems including ESC/Java2, RCC, Houdini, and the universes typechecker. It was originally developed by DEC-SRC but has been improved and extended over the past five years in a variety of ways.

Some MOBIUS subsystems under development use Javafe as their frontend, as it is a reliable, well-understood subsystem. Its main drawbacks are (1) that it only supports pre-Java 1.5 source code, thus does not support parameterized classes, enumerations, foreach loops, etc., and (2) its license is non-standard and not a recognized Free/Open Source license.

The Universes Typecheckers

The universes typesystems [MR07a] are used to describe ownership, containment, and aliasing properties of Java programs. The version of the universes typesystem described in Müller et al. [MR07b] is implemented in two subsystems in the Mobius PVE: in the core JML tool suite found in the JML2 Eclipse plugin, and in ESC/Java2.

FreeBoogie

FreeBoogie aims to be a backend for verification tools such as Spec$^\#$ and future versions of ESC/Java and similar tools. It takes as input a BoogiePL (Boogie Programming Language) file, generates verification conditions, calls provers such as Simplify and Z3, and reports the errors in a way that relates to the BoogiePL source. It is a work in progress.

Version 0.0 supports parsing, name resolution, flowgraphs, and typechecking, but no verification condition generation. Version 0.1 handles the first version of BoogiePL. Version 0.2 will handle the second version of the Boogie language. More information can be found on the FreeBoogie website.

The Spec$^\#$ system from Microsoft Research generates BoogiePL from the Spec$^\#$ language, which is a variant of Microsoft's C$^\#$ programming language with annotations. Another way of obtaining BoogiePL from a higher level language is via JACK [BBC$^+$07a], which translates Java with JML annotations into a variant of BML, and B2BPL, which was developed at ETH under the MOBIUS project and translates BML into BoogiePL. In fact, one design goal of BoogiePL was to be used as an intermediate language by multiple verification tools, thereby reducing the implementation effort needed to support multiple high-level languages and multiple static analyzes at the same time.

2.1.3 Static Analysis Subsystems using Annotation Generation

The Mobius PVE contains over a half dozen static analysis subsystems. Some of these static checkers were wholly developed within the MOBIUS project (e.g., ESC/Java2, as described in subsection 2.1.2), some that were developed by others, but evolved and extended within the project, and still others that were adopted and configured to complement the core analyses of MOBIUS. This section describes the second kind of static analysis subsystems—tools that MOBIUS has adopted and extended for our own purposes.

Race Condition Checker

The Race Condition Checker (RCC) was originally developed by researchers at the Digital Systems Research Lab (DEC-SRC) in the late 1990s [FF00]. Its source code was released to the general public along with the source for Simplify, Javafe, ESC/Java, and Houdini.

RCC's purpose is to (conservatively) statically check for the absence of race conditions via analysis of (type) annotated Java source code. As RCC's analysis is sound, if an annotated Java program passes RCC, then it is guaranteed to have no race conditions.

As the Mobius PVE is meant to eventually support reasoning about concurrent Java systems, determining whether a Java module is race-free is a critical first step in verifying any other properties. Unfortunately, RCC's source code release barely compiled, and was completely unmaintained.

Consequently, MOBIUS developers have "revived" RCC and released RCC version 2. This updated version of RCC incorporates the evolution of the Java language semantics, as expressed within the third revision of the JLS [GJSB05] and as realized in Java version 5, particularly with respect to the the Java Memory Model [Con04]. Moreover, several critical flaws in the original RCC type system and type checker were identified and corrected in this release.

Houdini

Like the Race Condition Checker (RCC), Houdini was also originally developed by researchers at the Digital Systems Research Lab (DEC-SRC) in the late 1990s [FL00]. Its source code was released to the general public along with the source for Simplify, Javafe, ESC/Java, and RCC.

Houdini automatically generates specifications written in JML for a Java module by statically analyzing the program code's structure. It has a number of built-in heuristics and performs some lightweight syntactic and semantic reasoning about method bodies to "guess" potential method preconditions, for example. These guesses are then fed to ESC/Java2 to see whether or not they are valid. If a specification seems to be valid (recall that ESC/Java is neither sound nor complete), then the specification is retained, otherwise it is discarded.

Houdini generates reports of its behavior in HTML. Houdini is also capable of taking a snapshot of its state for increasing the reliability of long runs (as running Houdini on a large code base can take hours, or even days). Finally, Houdini can run in a client-server mode, utilizing a (potentially larger, and more powerful) remote compute server.

2.1.4 Automated Theorem Prover Subsystems

Several new "backend" subsystems have been developed with MOBIUS resources. These systems are "backend" in the sense that they are used by many other subsystems and are deeply embedded in other tools, thus they are mainly interfaced via APIs and are heavily reused.

These subsystems include automated theorem provers, verification condition generators, specification generators, and intermediate representations for reasoning. All of these subsystems are used within the Mobius PVE, many for several purposes.

The Automated Prover API

The MOBIUS Automated Prover API provides a generic programming interface to automated theorem provers. The initial version of this API was designed by MOBIUS partners working in tandem with leaders in the SMT-LIB community. The intention is to have some future variant of this API be the standard API for all automated theorem provers, at least within the SMT-LIB community.

The API was developed by performing an analysis of the core operations provides by all major automated provers: prover initialization and shutdown, hinting to a prover about resource utilization bounds (primarily, time and space), declaration of new theories (via axiomatic definitions), making and retracting assumptions (as part of, e.g., a background predicate of a theory), and checking the validity of formulae.

The initial version of this API was designed with a formal specification written in JML, and example implementations were provided in Java, C, and OCaml. Communication with theorem provers is supported via pipes/sockets, native methods, and an XML-RPC-based messaging interface. This API is now used by ESC/Java2 and the CVC3 prover, and interfaces with the Z3 and Simplify theorem provers as well [BT07].

Simplify

The Simplify automated theorem prover was written by Greg Nelson and others at DEC-SRC around fifteen years ago. It is implemented in Modula-III. Until very recently it was still considered the premier automated theorem prover for software verification tools. It supports reasoning about a decidable fragment of first-order logic that includes a theory of arrays, uninterpreted function symbols, and linear arithmetic.

The only recent development on Simplify that has taken place is the MOBIUS port of it to Mac OS X, building new releases with new Modula-III compilers, as well as some simple bug fixes. There are no plans on doing any development on Simplify, as the world of first-order theorem provers has changed quite a bit over the past decade or so. Instead, we are working on integrating ESC/Java2 with new-generation SMT-LIB provers, as discussed later in this chapter.

Simplify has been wrapped in the new MOBIUS prover backend, thus is accessible in a simple, transparent fashion like many other new generation automated provers. Communication with Simplify is accomplished exclusively via piped data streams at this time. Additionally, if the prover terminates abnormally or becomes unavailable, it is restarted automatically. Finally, terminating a validity check via a timeout is also supported.

Fx7

Fx7 is an automated theorem prover implemented in the Nemerle programming language [ML06] by Michal Moskal, a visiting MOBIUS PhD student in 2007. Fx7 supports the same set of theories as Simplify and Z3, as well as a similar set of concrete syntaxes as Z3.

As Nemerle is implemented in Microsoft's .NET platform, the Mono runtime must be installed on non-Microsoft Windows Mobius PVE host operating systems. Currently the Fx7 prover is a hand-installed extension to the Mobius PVE. It will be included in a future release of the Mobius PVE, complete with a pre-packaged runtime for user convenience.

Fx7 has been "bridged" into the MOBIUS Automated Prover API in a fashion akin to the other provers discussed in this section. Piped datastreams are used to communicate with the prover. The API provides the ability to start and stop the prover, and control its resources use, and more.

In addition, because Fx7 is exclusively developed by MOBIUS-supported researchers, including contributors from UCD, various experiments in automated proving and verification are easy to carry out. In particular, Fx7 generates proofs of valid formulae and includes two proof checkers which run on embedded devices like PDAs [GM07]. These capabilities represent the first concrete experiences with proof generation and checking the Mobius PVE, albeit for extended static checking with ESC/Java2 rather than full functional verification.

Z3

The Z3 automated theorem prover is developed by Microsoft Research in Redmond [?]. It and Simplify are the primary theorem provers of the Boogie reasoning framework. Z3 is released in binary-only form. The most recent releases are only built for the Microsoft Windows operating system. An older, out-of-date release was built for Linux to conform to the SMT-LIB competition's rules.

Z3 is a very powerful modern prover, thus it has been "bridged" into the MOBIUS Prover API as well. It supports the Simplify and SMT-LIB concrete syntaxes, as well as its own.

Whereas Z3 only runs on the Microsoft Windows platform, the Mobius PVE runs on all modern platforms, including Apple's OS X and UNIX variants like Linux, Solaris, etc. We wish to support as many modern theorem provers as possible within the Mobius PVE, but supporting Z3 on non-Windows systems is a non-trivial task.

Recently Z3 has been tested running within the a Wine runtime [win08] on Intel-based systems running Linux. This mechanism (spawning Z3 within Wine, communication via piped datastreams, etc.) is now built into the MOBIUS Automated Prover API.

In theory, such a mechanism will work on all Intel-based Wine runtimes, like those available for other UNIX variants and OS X. In addition, we plan on experimenting with executing Z3 within a virtualized Microsoft Windows installation (e.g., using Parallels or VMware, both of which are used for Mobius PVE development and testing) on OS X on Intel.

CVC3

The CVC3 automated theorem prover is currently developed by two teams, one located at the University of Iowa and the other at New York University. It comes from a legacy of SMT prover developed at Stanford University with the SVC system.

The CVC3 prover uses a version first-order logic with polymorphic types and quantifiers. It has powerful built-in theories: it has rational and integer arithmetic, arrays and other alike data structures as well as inductive data types and equality over uninterpreted function symbols.

Lately, the CVC3 prover has been tightly integrated in ESC/Java2's prover backend, using its brand new Java API. This version of CVC3 is the version that is included in the Mobius PVE Epsilon release. Even being a so-called unstable version it is stable enough to be properly used in ESC/Java2.

Yices

Yices is an automated theorem prover that is being developed by SRI International. It is an SMT solver. It provides counterexamples and unsatisfiable cores. It has numerous theories tightly integrated, and it supports user constraints as well.

The main algorithm it uses is similar to the one in Simplify. It handles uninterpreted function symbols and equalities. It can handle the solving of MaxSMT problems as well as SAT and MaxSAT kind of problems.

It is used in the interactive theorem prover PVS, as a decision procedure, as well as in the Symbolic Analysis Laboratory (SAL) tool suite. SAL is a tool made to help analyse concurrent systems if they are specified as transition relations.

2.1.5 Prover Editor Subsystems

The MOBIUS Prover Editor [JC08] is an integrated frontend for interactive theorem provers, currently the Coq prover, that is used to prove the generated verification conditions interactively. Originally developed for JACK, it has now been integrated in the MOBIUS environment. See [BBC$^+$07b] for more information.

It is composed of several components, a core surrounded by several plugins specific to provers.

Prover Editor's Core Component

The basic idea behind the design of the MOBIUS Prover Editor is that reading and writing large-scale formal specifications is much the same as reading and writing large-scale program source [Kin04]. Common programming interface idioms like outline views, syntax highlighting, (possibly type-aware) identifier completion, file and construct templates, etc. are naturally adopted and adapted to support development of, and reasoning about, formal models expressed in higher-order provers.

Additionally, the human-computer interface between the user and the prover via the Mobius PVE must not only be simple, but most importantly, familiar to traditional interactive prover users. Thus, most standard key bindings for the Emacs-based front-end are available from within the MOBIUS Prover Editor.

While the MOBIUS Prover Editor primarily supports Coq, development is underway to also support the PVS theorem prover, porting the improvements previously identified by Kiniry and Owre [KO03] and reemphasizing the Prover Editor's prover-generic nature.

The Coq Language Support

There is an extended support of the Coq language in ProverEditor. It is composed of two parts, one offering basic interaction with the theorem prover, and the other providing a high level API permitting to send commands and interpret the output of Coq directly in Java. This interaction process is extensively described in [JC08].

2.1.6 Core Application Programmer Interfaces Subsystems

The subsystems described in this section are subsystems that were made in order to support other MOBIUS tools, for their integration inside of Eclipse as well as enabling better profiling of their performances.

The Rich Client Platform Plugin and Library

In order to make the PVE a fully integrated application inside of Eclipse, two things were necessary: establishing a base library of standard functions used for every MOBIUS graphical plugin, and providing an extended integration as a Rich Client Platform. This come in the form of two subsystems, the pluginlib providing facilities for graphical interaction and the PVE plugin, gathering all the other MOBIUS plugins in order to build an autonomous application out of them.

The Logging API

The MOBIUS Logging subsystem (or MobLog for short) is an API used by programmers to make assertions, and emit log messages, about a software system. At its core, it combines Java and JML's assert statements with functionality akin to that found in the Java logging framework (via the java.util.logging package) and the Apache Foundation's log4j logging service.

The original design for the Logging subsystem predates not only the MOBIUS project, but the existence of Java's logging package and log4j [Kin98]. But, even though this framework is about a decade old, it still has several features not found in log4j or similar frameworks including first-class logging contexts, integrated statistics collection, logging categories, and more.

Within this project, it has been updated and extended for incorporation into the Mobius PVE for several reasons.

First, as it was designed and written with formal specifications (initially written in Jass, now in JML), it represents a case study in verification and thus will be used to test various other MOBIUS subsystems. All classes are annotated with lightweight JML assertions, and a high-level BON [WN95] specification is also included with the release. Moreover, the entire software system has been updated and analyzed within the Mobius PVE itself, passing all static checkers, including ESC/Java2.

Additionally, the MobLog framework supports complex concurrent and distributed systems' development, like that found in the Mobius PVE. Tracing the execution of, and understanding the behavior of, multiple concurrent threads requires a thread-aware logging framework.

Finally, its availability under MOBIUS control provides an opportunity to unify the several disparate logging subsystems found within other MOBIUS components. Thus, it is our aim that all Mobius PVE logging, tracing, and statistics-gathering is performed via this single unified subsystem.

2.1.7 Java Bytecode Subsystems

One of the end goals of the MOBIUS project is the production of PCC certificates for Java. To accomplish this goal, reading, writing, manipulating, specifying, and reasoning about Java bytecode is mandatory. This section discusses the initial subsystems developed for the Mobius PVE to support these objectives.

The Umbra Bytecode Editor

Umbra is an editor for bytecode files which supports editing of BML specifications. This editor can manipulate bytecode files in a textual form. It can also add, change, and remove instruction mnemonics so that the actual functionality encoded in the class file methods changes.

In addition to the modification of bytecode programs, Umbra supports viewing, adding, deleting and modifying specifications written in the Bytecode Modeling Language (BML), a counterpart of JML at the bytecode level.

A separate part of the Umbra editor is a library called BMLlib. BMLlib parses fragments of BML specifications, prints them out in the aforementioned textual format, and reads and writes BML specifications in class files. More details are presented in [SCB+08].

JML2BML Compiler

The Umbra editor is also equipped with an automatic tool to transform the JML specifications to BML ones called JML2BML. The compiler can also work as a standalone tool which takes source code with JML annotations, a class file and adds the BML specifications to the class file. More details are presented in [FJS08].

BMLVCgen Condition Generator

The bytecode subsystem is able to generate the verification conditions to be proved in Coq. The tool which does that, BMLVCgen, uses the mechanisms of the MOBIUS Direct Verification Generator so the resulting conditions are very similar to the ones obtained from that. This makes it possible to conduct direct verification of bytecode programs when no source code is available. More details are mentioned in [CHS09].

Certificate Generation

The bytecode tool set contains also a small tool CCT which enables the possibility to pack a certificate being e.g. a proof done in Coq to class files. This tool also makes possible to unpack and check that the code in the class file indeed has the property proven by the certificate. More details are mentioned in [CHS09].

2.1.8 DirectVCGen Subsystems

One of the aim of the MOBIUS project was to implement a Proof Carrying Code framework.

The Bicolano Plugin

Bicolano [Pic06] is a Coq library that formalizes the memory model of Java, as well as the bytecode instructions and their semantics. To translate Java classfiles into this Coq formalization, a tool called Bico has been developed within MOBIUS. This is joint work of INRIA and Warsaw University [mC07a].

The Bicolano plugin is a package to make Bicolano handily useable by other subsystems, like the DirectVCGen, BMLVCGen, Bico or Bico$^+$.

Bico and Bico$^+$

For each classfile, Bico generates a type, signature, and implementation formalized in Coq. This generation is modular. Bico also generates summary files to simplify the use of the definitions.

The main limitation of Bico is due to the size of the Java system library. The classes contained in this library are usually not generated, because the resolving of the dependencies would make Bico generate the Coq formalisation for all the system libraries (more than 200 classes). This is not a problem by itself, but loading the generated files into Coq uses a large amount of time and memory. Therefore, Bico is most useful when applied on user-written class files, where this problem tends not to arise.

Bico$^+$ is a tool built upon the top of Bico which permits to generate annotations expressed in the Bicolano formalisation. As an input it takes annotations specified in a data structure. It is used in two fashions: first in the DirectVCGen, with the annotations being translated from JML, second in BMLVCGen with the annotations being translated from BML.

The Direct Verification Condition Generator (DirectVCGen)

The MOBIUS DirectVCGen is a tool built upon Bico, Bicolano, the ESC/Java2 parser, as well as Mobius PVE's prover backend. It is built around two verification condition generators, one over source code and one over bytecode.

The MOBIUS DirectVCGen is called "direct" because, contrary to the other verification condition generators of the Mobius PVE, it uses a simple weakest precondition calculi without any use of an intermediate language. The source VCGen is written in Java and the bytecode VCGen is written in Coq. It is part of the proof transforming compiler scheme, as explained in the deliverable 4.3 [mC07b].

One of the main goals of this tool is to translate the proofs for the verification conditions generated from the source code to proofs for the verification conditions generated from the bytecode. To allow this transformation, both VCGen subsystems have to be reasonably similar. Both weakest precondition calculi have exactly the same set of rules. The only thing that differs between the two calculi is their way of handling variables. Of course, bytecode variables can be easily programatically deduced from the source code ones.

The background predicates for both verification conditions are found in the same theory, that of Bicolano. The tool Bico is used to translate a program into the Coq formalization. The backend for the VCGen over source code is a specific Coq backend that uses Bicolano as its memory model.

The mechanisms of the MOBIUS DirectVCGen are used by the bytecode subsystem to generate verification conditions based on the bytecode alone and BML specifications. The transition between bytecode combined with BML annotations and the DirectVCGen is done by a tool called BMLVCGen.

DirectVCGenUI

The user interface of the DirectVCGen is a simple integration inside of Eclipse to seamlessly be able to see the status of the verification conditions (if they have been proved or not). This integration is minimalist, and evolves around a simple panel, which uses other components of the Mobius PVE, like the MOBIUS perspective and ProverEditor.

This concludes a summary of Mobius PVE subsystems developed under the MOBIUS project.

2.2 Subsystems Used within MOBIUS but Developed Elsewhere

Currently eight major "external" subsystems are included in the Mobius PVE. All of these subsystems have programmatic interfaces available to Mobius PVE developers and are wrapped in Eclipse plugins for Mobius PVE users. Many of them are also "tuned" for verification, as described in the following pages.

These particular tools have been chosen for inclusion in the Mobius PVE because they have been very useful to MOBIUS partners in verification-centric software engineering over the past several years [KZ08]. As new tools appear in the Eclipse community, they are objectively evaluated by MOBIUS partner UCD for inclusion in future releases of the Mobius PVE.

2.2.1 Mylyn

Mylyn is a "Task-Focused Interface" plugin developed originally by researchers at the University of British Columbia, and is now a full-fledged Open Source project recognized by the Eclipse Foundation. Mylyn makes tasks first-class entities within Eclipse and supports offline, asynchronous interactions with the MOBIUS Trac bug and feature tracker from within Eclipse, and thus from within the Mobius PVE.

Additionally, when a task is "active", Mylyn watches the Mobius PVE user work and figures out what constructs (files, classes, methods, theories, definitions, lemmas, etc.) are important with respect to that particular task. This means that only those constructs relevant to the current work, be it developing new software or performing a verification, are visible at any point in time. When a task is labeled as "inactive", Mylyn automatically snapshots the current state of the Mobius PVE and puts everything away so the user is returned to a clean initial desktop. Furthermore, this context can be saved as a file or attached to a Trac wiki page or ticket so that others may return to the same state as the original Mobius PVE user.

2.2.2 Version Control

MOBIUS developers primarily use two version control systems: CVS and Subversion. Eclipse comes with a CVS plugin by default. Two primary Subversion plugins are available today: Subclipse and Subversive. After evaluating both options, we chose to ship Subclipse with the Mobius PVE, as it is more reliable and has a more appropriate feature set for dependable systems development. No special configuration of our version control plugins has taken place.

The Eclipse Mobius PVE developer workspace that we provide via the MOBIUS Trac server is pre-configured and points (anonymously) to all of the version control repositories relevant to MOBIUS work, including those hosted by MOBIUS partners, and those hosted by external sites like SourceForge and our North American colleagues.

The student/instructional/demonstration Mobius PVE workspace that we provide does not point to any version control repositories.

2.2.3 CheckStyle

CheckStyle is a plugin that performs source code style checking. Style checking involves statically analyzing both documentation and source code against simple, syntactic rules. At this time CheckStyle does not check JML annotations. Thus, beyond providing example coding styles codified for checking with this plugin, no special configuration of the CheckStyle plugin has taken place.

As described in chapter 4, a major part of the MOBIUS process is ensuring that source code is of a certain quality, measured in terms of the number of errors and warnings produced by several static checkers working in tandem. The first of these checkers is CheckStyle.

The Mobius PVE comes with several example coding styles: a basic style that promotes readable code, a moderate style developed by MOBIUS partner Warsaw University that focuses on dependable code, and a rigorous style developed by MOBIUS partner University College Dublin that focuses on verifiable code. Additionally, all non-basic styles inherit from the basic style, thus all checks in the basic style are also contained in the moderate and rigorous styles.

Style checkers work hand-in-hand with code generators. In Eclipse, there are three basic kinds of code generators: code templates and skeletons, the results of refactoring, and writing new code "by-hand" while using Eclipse's built-in code formatter.

Consequently, each of these mechanisms for generating code must be "tuned" to match the expected structures of the tool that checks code. As such, the Mobius PVE comes with appropriate code templates and a code formatting specifications to match each coding style. Refactoring need not be "tuned" as refactored code is automatically styled according to the active code formatting style.

In summary, the Mobius PVE assists developers in writing clear, readable, maintainable documentation and program code by both generating code to a particular per-project style and checking that code conforms to said style.

The next level of rigor in static checking uses not just syntactic checks, but some lightweight semantic checks as well. The two main plugins used in the Mobius PVE for such are FindBugs and PMD, covered in the following two sections.

2.2.4 FindBugs

Static checkers like FindBugs check for programming problems beyond syntactic code style checking. FindBugs is a configurable bytecode-level checker that analyzes code for common programming errors. FindBugs does not use a formal semantics or a theorem prover; its checks are entirely based upon the structure of Java bytecode.

In the Mobius PVE, we have "tuned" FindBugs to perform those checks that complement and support writing code which is verifiable using MOBIUS verification tools. In other words, if a program passes the Mobius PVE pre-configured static checkers, it is more amenable to formal verification.

Each check that is enabled in FindBugs has been evaluated and objectively analyzed to determine if it should be enabled or disabled for verification-centric programming. This evaluation is ongoing as new checks become available as FindBugs evolves.

2.2.5 PMD

Likewise, we have "tuned" the static checker PMD in a similar fashion. PMD analyzes source code, rather than bytecode. Thus, it is not usable in all circumstances, as sometimes source is not available due to licensing, legal, or intellectual property-related reasons.

Evaluation and maintaining the configuration of PMD in the Mobius PVE mirrors that of FindBugs—as the plugin evolves, so does the set of checks enabled by default to support verification-centric development.

2.2.6 Metrics

A variety of plugins support measuring a suite of standard software engineering metrics in program code. Years of experience with automatic and interactive verification have taught Mobius PVE developers that, code which has a specific range of values for a set of well-understood metrics are indicative of program code that is easier to verify. Therefore, providing a plugin that performs metric static analysis is useful to Mobius PVE users.

Two metrics-related plugins ship with the current "epsilon" version of the Mobius PVE: NCSS and Metrics.

NCSS

JavaNCSS is a simple program that performs some very basic analysis of program source code. In particular, JavaNCSS counts the Non-Commenting Source Statements (NCSS) of Java source at the module and method level, as well as summarizing a variety of other tertiary data.

The NCSS plugin included with the Mobius PVE provides an extremely simple interface to triggering JavaNCSS from within Eclipse. Analysis is only supported at the project level, and the output of said analysis is provided within a read-only window containing the raw text output of JavaNCSS.

As we gain more experience with and "tune" the Metrics plugin discussed in the next section, it is expected that future versions of the Mobius PVE will not include the NCSS plugin.

Metrics

The Metrics plugin, in addition to performing all of the actions that NCSS does, also is capable of analyzing source code for well-known metrics found in the software engineering literature. Also, metrics at the class and method level are summarized in a more useful contextual structured fashion, rather than in a plain text window. For example, when a given code construct is selected with the mouse, the Metrics reporting window automatically updates to reflect the metrics of that construct.

MOBIUS partners have not gained much hands-on experience in some of these more advanced kinds of metrics, like various kinds of code coupling analyses. Thus, there is little concrete evidence about the verifiability of code with respect to such metrics. In fact, as there is little scientific published data on such, the integration of verification and metrics tools in the Mobius PVE provides the foundation for new research in such.

2.2.7 EclEmma

The following external plugin integrated into the Mobius PVE focuses on code coverage analysis. The code coverage plugin chosen for use within the Mobius PVE is called EclEmma.

Emma is a code coverage analysis tool that, during program execution (typically during unit testing), automatically instruments program bytecode to gather information about various kinds of standard code coverage measures, including branch and statement coverage. Emma reports such analysis back to the programmer through a series of reports.

EclEmma wraps Emma into an Eclipse plugin with a rich user interface. Coverage data is summarized to the user in a simple graphical fashion where program source is highlighted in various ways to indicate if a given construct has been executed during a given test run. For example, if a particular method was never executed during unit testing, its body is highlighted in red.

No customization has yet taken place for EclEmma. The main customization requested thus far is the ability to perform code coverage analysis during runtime assertion checking. Highlighting the program source code and executable specifications (in particular, method contracts and invariants) executed in a given test run will be particularly useful to Mobius PVE users. Such data will help users understand the completeness of their specifications with respect to their program code, as well as the completeness of their specifications with respect to their unit tests.

Additionally, given EclEmma performs aggregate coverage analysis, e.g., a given class's executable statements are 90% covered by the current unit tests. Such data will give a concrete measure that can be related to other aggregate measure like those reported by the variety of static analysis tools included in the Mobius PVE like FindBugs and ESC/Java2. A verification "dashboard" is planned that succinctly, visually, interactively summarizes all of this information for the programmer and a development team.

2.2.8 BCEL

BCEL is the Java library that has been chosen in the MOBIUS project to inspect and manipulate bytecode. It is used mainly in the BML subsystems, the DirectVCGen subsystems and the ESC/Java2 subsystem.

Its main use is for embedding annotations in the BML subsystems, and in DirectVCGen and ESC/Java2 it is used to inspect the structure of the class hierarchy especially when no source code for a specific class is provided. It is packed as a plugin for Eclipse, and all the libraries are exposed through it to the other MOBIUS tools.

2.2.9 BONc

BON is a textual and graphical domain-independent specication language akin to UML, but which focuses on all software engineering stages from domain analysis to architecture to contract-centric component design.

BONc is a parser, typechecker and documentation generator for BON. It primarily focuses on checking the correctness of BON documentation, as well as automating some tasks that are performed during development with BON. More information can be found on the BONc website.

2.2.10 Beetlz

Beetlz is a tool providing automated support for consistency checking between a detailed BON model and the JML specification and Java implementation. It delegates parsing of the BON and Java/JML to BONc and OpenJML respectively, and consumes the structures produced by these tools during parsing.

Discrepancies between related artifacts are detected and presented in a cohesive manner. Beetlz also provides support for Java/JML code generation as well as BON model extraction. Developers can also provide specific knowledge to guide the tool with facts and refinement relations that were not automatically deducible. More information can be found on the Beetlz website.

This concludes a summary of Mobius PVE subsystems developed by parties outside of the MOBIUS project. Furthermore, this concludes a summary of all Mobius PVE subsystems.

Chapter 3

Using the MOBIUS Program Verification Environment

As the Mobius PVE is simply an enriched version of Eclipse, Eclipse users will find the Mobius PVE quite familiar. This chapter provides a high-level developer walk-through of the configuration and use of the Mobius PVE.

Screenshots included in this deliverable are of the current "epsilon" version of the Mobius PVE. The final "epsilon" version has more consistent, tighter integration between subsystems, but as it is being shipped after this deliverable is submitted, we do not include screenshots from it here. The Mobius Trac http://mobius.ucd.ie/, which contains a constantly-updated version of this deliverable as user documentation will contain up-to-date screenshots when appropriate.

3.1 General Walk-through of Using the Mobius PVE

The first step in using the Mobius PVE is to obtain and install it from the MOBIUS website.

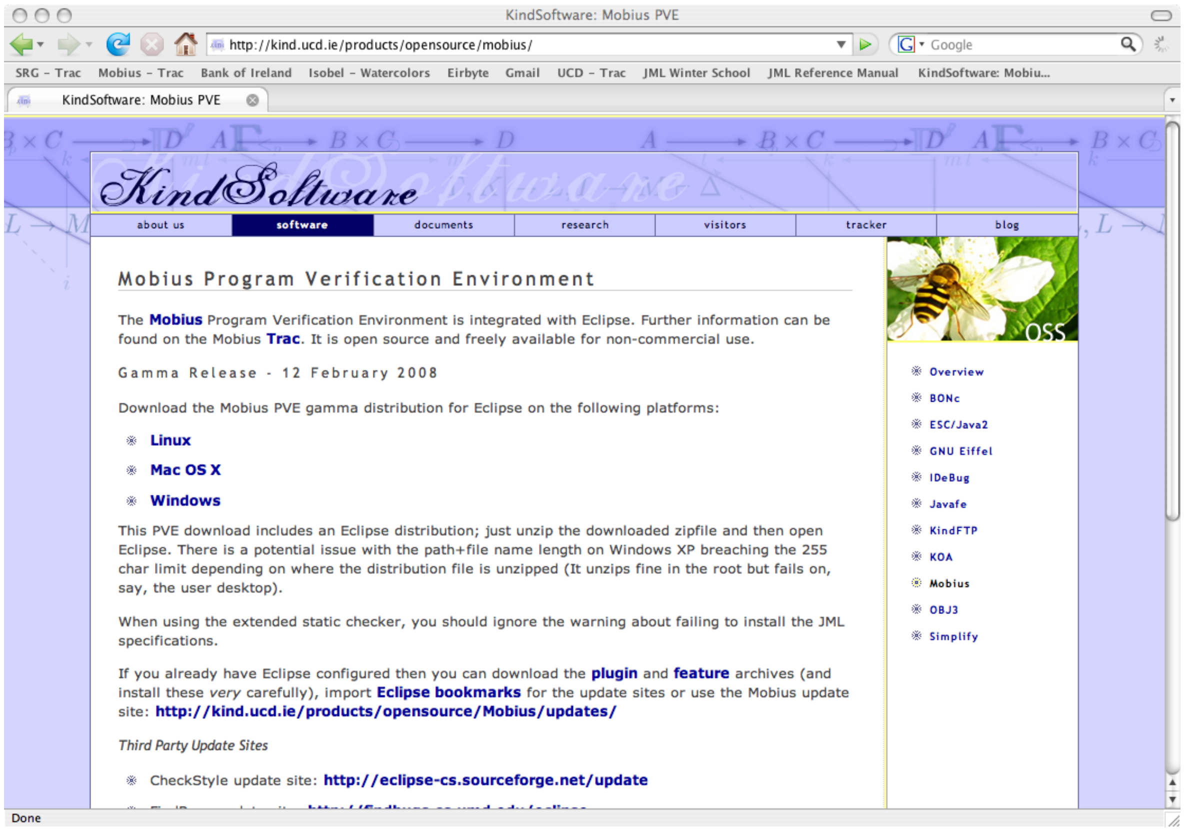
3.1.1 Downloading

The Mobius PVE is available from the MOBIUS Trac server http://mobius.ucd.ie/ and via the KindSoftware research group website (the research group of the MOBIUS UCD partner), as seen in Figure 3.1. Pre-configured Eclipse builds are available for all main platforms including Microsoft Windows, Apple's Mac OS X, and Linux. Download the appropriate archive; it can be unpacked anywhere on the local disk.

Sample Workspace A sample Eclipse workspace is also available from the aforementioned sites. This workspace includes example projects and documentation about the Mobius PVE.

3.1.2 Configuring a Java Project for Verification

Each main subsystems included in the "epsilon" release of the Mobius PVE is configured separately. First one must create a new Java project, or import an existing Java project. One may also simply use an existing Eclipse workspace with the Mobius PVE.



Mobius Program Verification Environment

The **Mobius** Program Verification Environment is integrated with Eclipse. Further information can be found on the Mobius **Trac**. It is open source and freely available for non-commercial use.

Gamma Release - 12 February 2008

Download the Mobius PVE gamma distribution for Eclipse on the following platforms:

- **Linux**
- **Mac OS X**
- **Windows**

This PVE download includes an Eclipse distribution; just unzip the downloaded zipfile and then open Eclipse. There is a potential issue with the path+file name length on Windows XP breaching the 255 char limit depending on where the distribution file is unzipped (It unzips fine in the root but fails on, say, the user desktop).

When using the extended static checker, you should ignore the warning about failing to install the JML specifications.

If you already have Eclipse configured then you can download the **plugin** and **feature** archives (and install these *very* carefully), import **Eclipse bookmarks** for the update sites or use the Mobius update site: **http://kind.ucd.ie/products/opensource/Mobius/updates/**

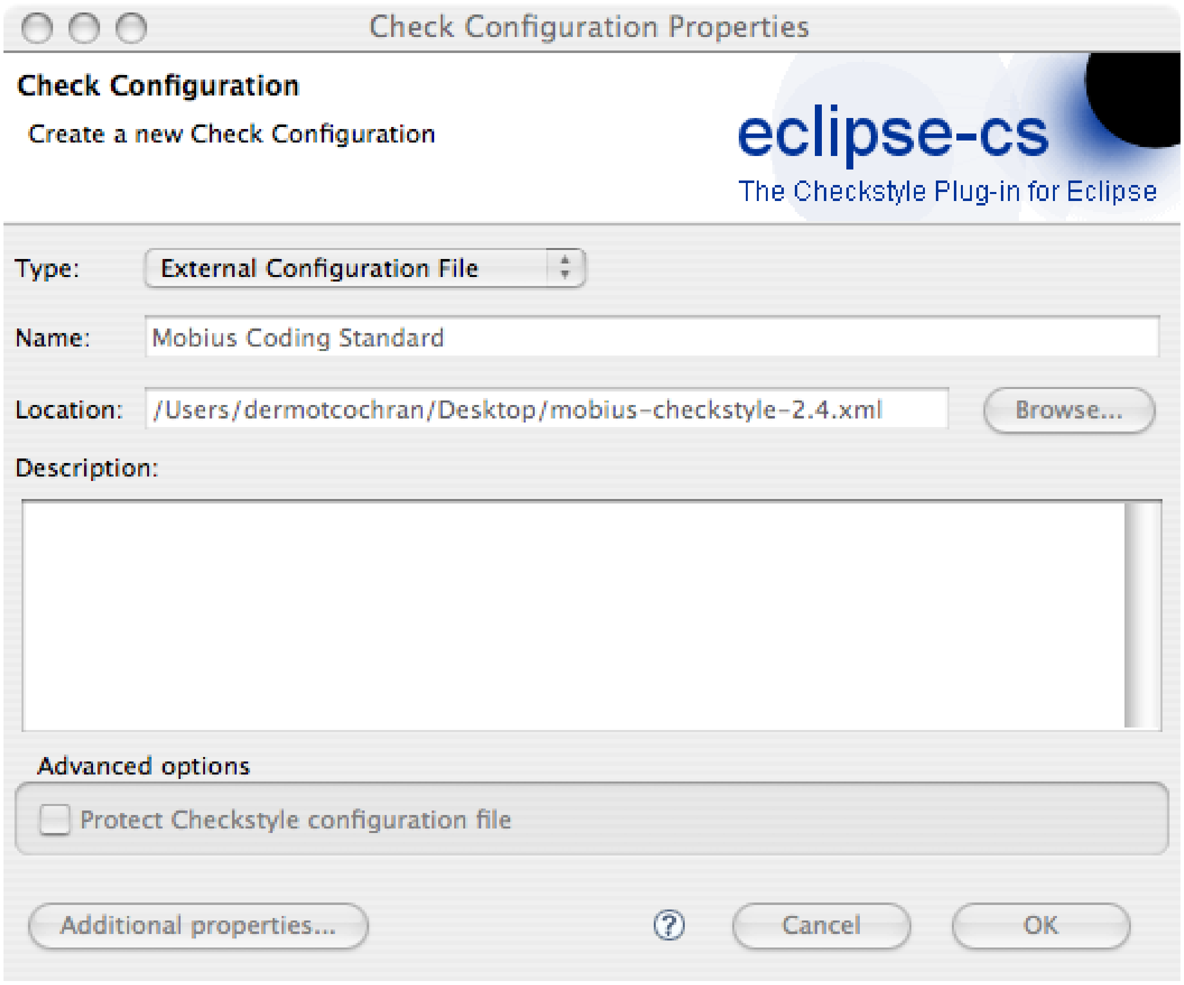
Third Party Update Sites

- CheckStyle update site: **http://eclipse-cs.sourceforge.net/update**

Figure 3.1: The Mobius PVE Download Site at KindSoftware.

Coding Standards

The first subsystem that one should configure is the CheckStyle coding standard checker and the complementary Eclipse code formatter. The CheckStyle plugin is configured via the Project Properties dialog, available either via the Project menu or via a right mouse action on the project name in the resources browser. The Mobius PVE comes pre-configured with the MOBIUS coding standard built-in.


Check Configuration Properties
Check Configuration
Create a new Check Configuration
eclipse-cs
The Checkstyle Plug-in for Eclipse
Browse...
Advanced options
Additional properties...
Cancel
OK

Type: External Configuration File

Name: Mobius Coding Standard

Location: /Users/dermotcochran/Desktop/mobius-checkstyle-2.4.xml

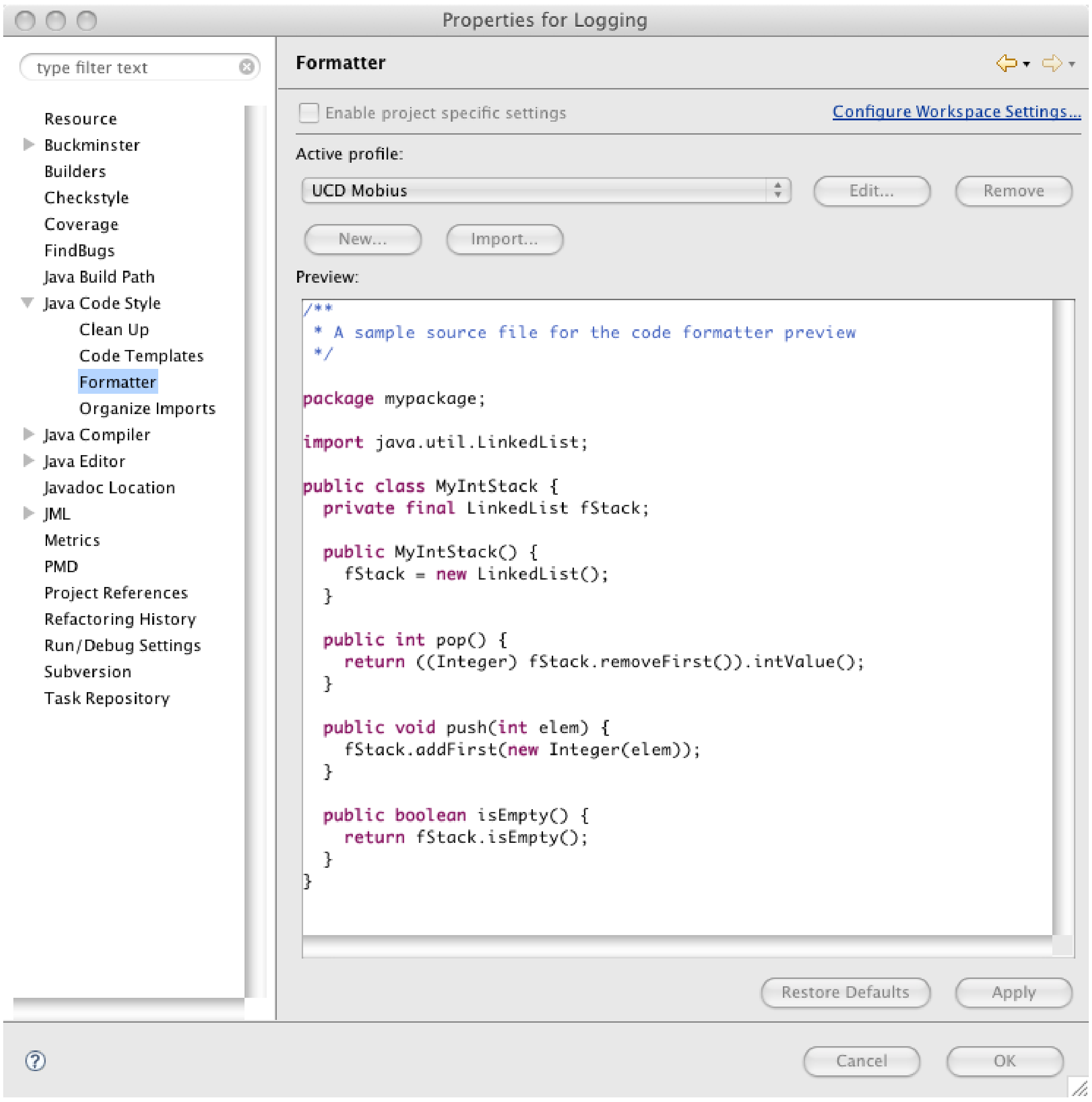
Description:

Protect Checkstyle configuration file

Figure 3.2: Importing a new coding standard.

If one wishes to further configure the default coding standard, either the configuration can be tweaked via the plugin's preferences panel, or one can import, duplicate and edit the external MOBIUS coding standard configuration file, available from the MOBIUS Trac server. Figure 3.2 illustrates this possibility.

Configuring the Eclipse code formatter takes place in a similar fashion. The Mobius PVE is pre-configured with several code formatters. Figure 3.3 shows the example UCD code formatter configuration that is included with the sample workspace.



Enable project specific settings

Active profile: UCD Mobius

```
/**
 * A sample source file for the code formatter preview
 */

package mypackage;

import java.util.LinkedList;

public class MyIntStack {
  private final LinkedList fStack;

  public MyIntStack() {
    fStack = new LinkedList();
  }

  public int pop() {
    return ((Integer) fStack.removeFirst()).intValue();
  }

  public void push(int elem) {
    fStack.addFirst(new Integer(elem));
  }

  public boolean isEmpty() {
    return fStack.isEmpty();
  }
}
```

Figure 3.3: Configuring the Eclipse Code Formatter.

Configuring and Using the JML Plugin

The JML plugin is configured on a per-project basis. The JML typechecker, compiler, and runtime assertion checker are configured independently. Figure 3.4 shows the preferences panel for configuring the JML typechecker.

JML typechecking or compilation is currently accomplished via a menu action, as seen in Figure 3.5. In a future version of the Mobius PVE, autobuilders will be intelligently triggered instead.

Configuring and Using Static Checkers

Several static checkers are included in the Mobius PVE. First, configuring and executing the main static checker, ESC/Java2, is summarized.

ESC/Java2 The main static checker included with the Mobius PVE at this time is ESC/Java2. ESC/Java2 is configured via menu actions and preferences panels.

To initially configure the plugin, one must enable the ESC/Java "nature" (a set of configurations that ensures the plugin is properly configured for use) and "builder" (an action that is triggered at specified times, typically to build a system or to perform some analysis). The JML menu contains item to perform initial setup of the plugin (Figure 3.6).

Project-specific configuration of ESC/Java2 is accomplished via a Project Properties panel, as seen in Figure 3.7. Via this control panel, one controls all of the standard warnings for ESC/Java2, the use of various theorem provers, and programming language variants, including JavaCard (see Figure 3.8).

The ESC/Java builder can be enabled to run each and every time a Java source file is saved. This is accomplished via another item in the JML menu, as seen in Figure 3.9.

Finally, ESC/Java2 can be manually executed on any project subunit (file, package, etc.) via a right mouse action on the appropriate resource (Figure 3.10).

Other Static Checkers Configuring plugins like PMD, FindBugs, Mylyn, EclEmma and Metrics all happen in the standard fashion. Decide which, if any, of the non-MOBIUS static checkers should be enabled as auto-builders. (An auto-builder is a builder that is automatically triggered when a specific action takes place, e.g., when a file is saved.)

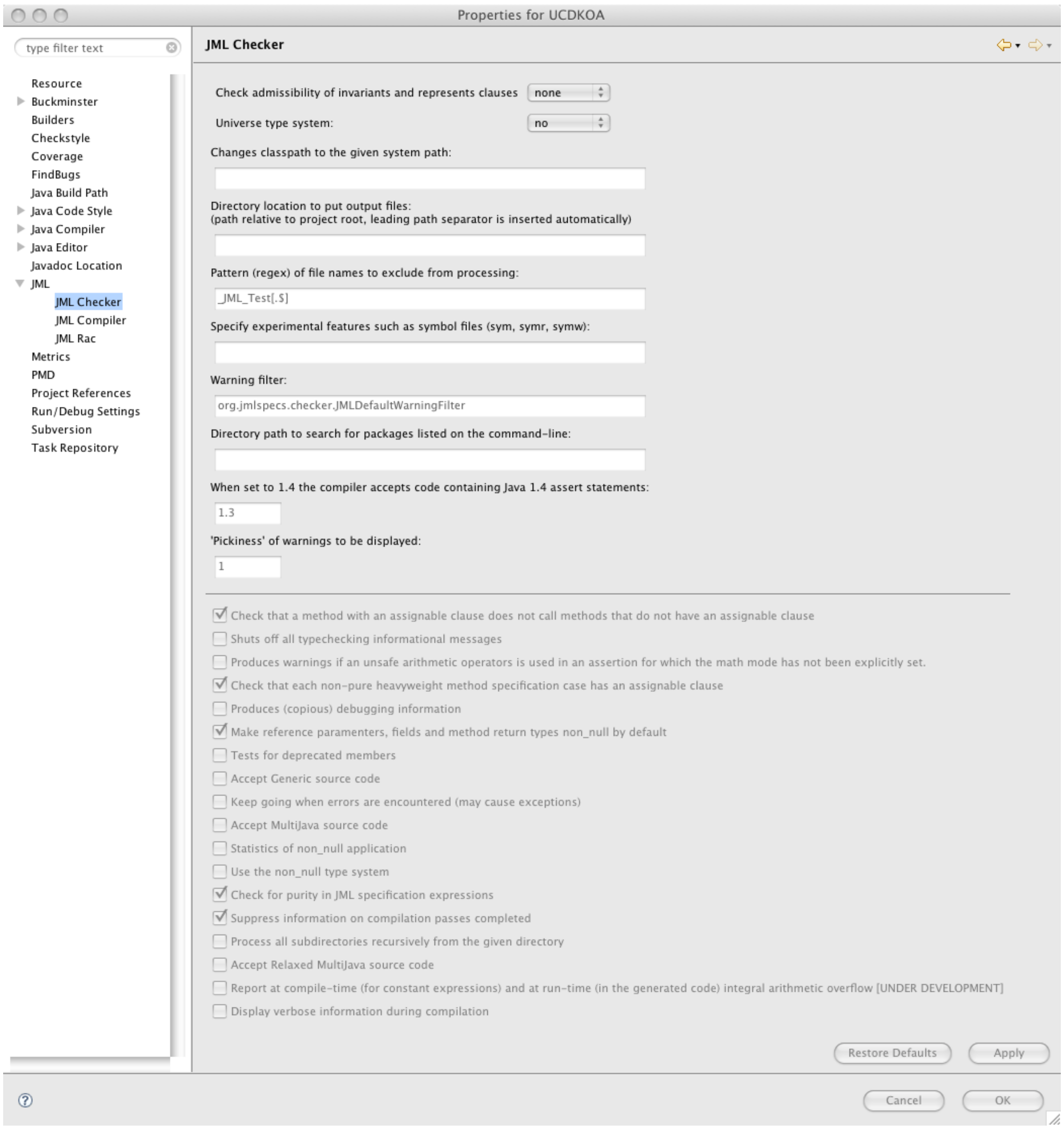
As all plugins are pre-configured to compliment verification with MOBIUS subsystems, their preference settings should, in general, be untouched.

Configuring Other Subsystems

Several other subsystems (the JML Folding subsystem, the MOBIUS Coq editor, the Umbra bytecode viewer, etc.) are configured in similar ways, but are not discussed here for brevity's sake.

3.1.3 Performing Verification

Performing verification within the Mobius PVE is integrated directly into the standard practices of software development.



JML Checker

Check admissibility of invariants and represents clauses: none

Universe type system: no

Changes classpath to the given system path:

Directory location to put output files:
(path relative to project root, leading path separator is inserted automatically)

Pattern (regex) of file names to exclude from processing: _JML_Test[.$]

Specify experimental features such as symbol files (sym, symr, symw):

Warning filter: org.jmlspecs.checker.JMLDefaultWarningFilter

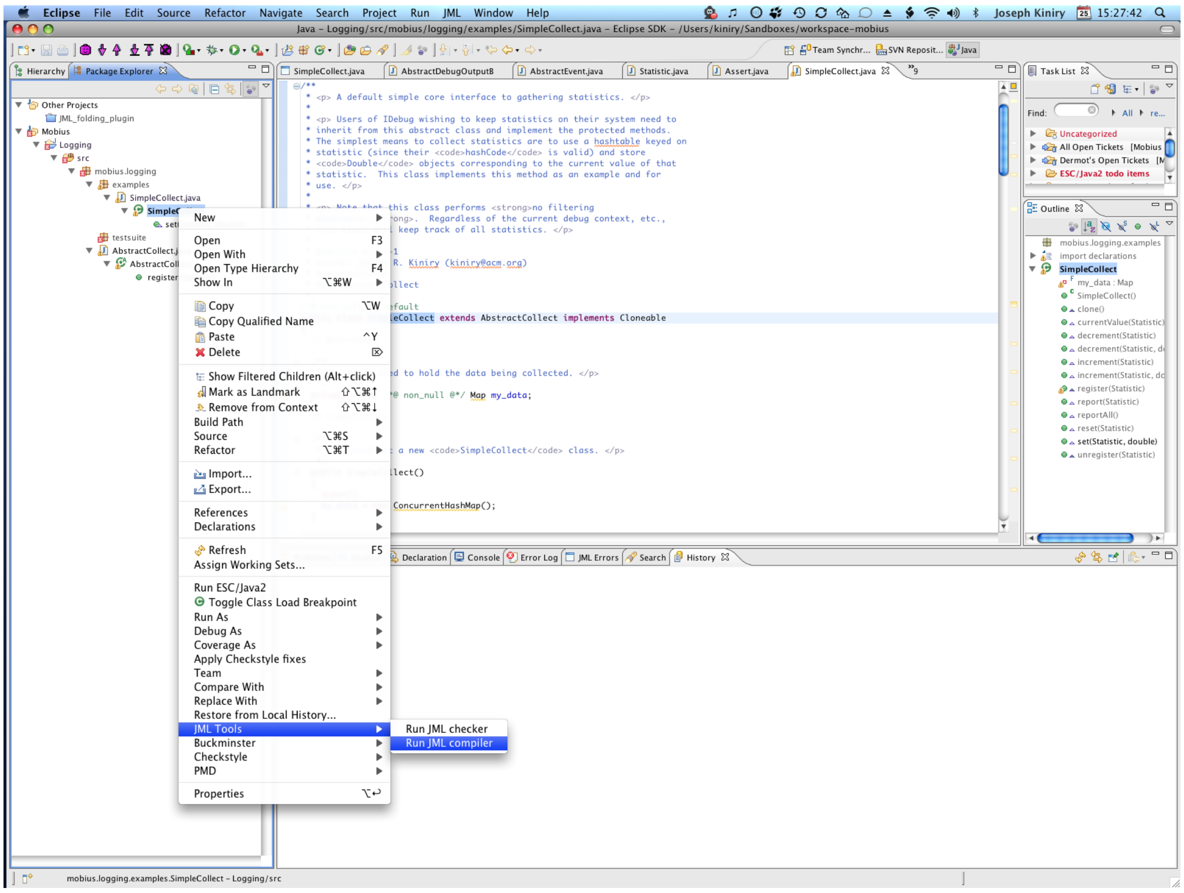
Directory path to search for packages listed on the command-line:

When set to 1.4 the compiler accepts code containing Java 1.4 assert statements: 1.3

'Pickiness' of warnings to be displayed: 1

- [✓] Check that a method with an assignable clause does not call methods that do not have an assignable clause
- [ ] Shuts off all typechecking informational messages
- [ ] Produces warnings if an unsafe arithmetic operators is used in an assertion for which the math mode has not been explicitly set.
- [✓] Check that each non-pure heavyweight method specification case has an assignable clause
- [ ] Produces (copious) debugging information
- [✓] Make reference paramenters, fields and method return types non_null by default
- [ ] Tests for deprecated members
- [ ] Accept Generic source code
- [ ] Keep going when errors are encountered (may cause exceptions)
- [ ] Accept MultiJava source code
- [ ] Statistics of non_null application
- [ ] Use the non_null type system
- [✓] Check for purity in JML specification expressions
- [✓] Suppress information on compilation passes completed
- [ ] Process all subdirectories recursively from the given directory
- [ ] Accept Relaxed MultiJava source code
- [ ] Report at compile-time (for constant expressions) and at run-time (in the generated code) integral arithmetic overflow [UNDER DEVELOPMENT]
- [ ] Display verbose information during compilation

Figure 3.4: Configuring the JML Checkers.



```
/**
 * <p> A default simple core interface to gathering statistics. </p>
 *
 * <p> Users of IDebug wishing to keep statistics on their system need to
 * inherit from this abstract class and implement the protected methods.
 * The simplest means to collect statistics are to use a hashtable keyed on
 * statistic (since their <code>hashCode</code> is valid) and store
 * <code>Double</code> objects corresponding to the current value of that
 * statistic.  This class implements this method as an example and for
 * use. </p>
 *
 * <p> Note that this class performs <strong>no filtering
```

Figure 3.5: Executing the JML Compiler.

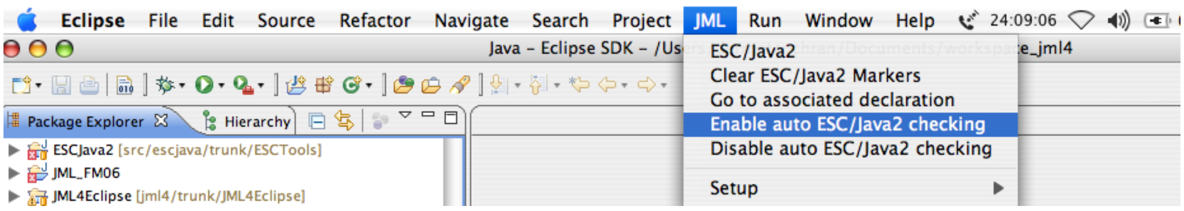
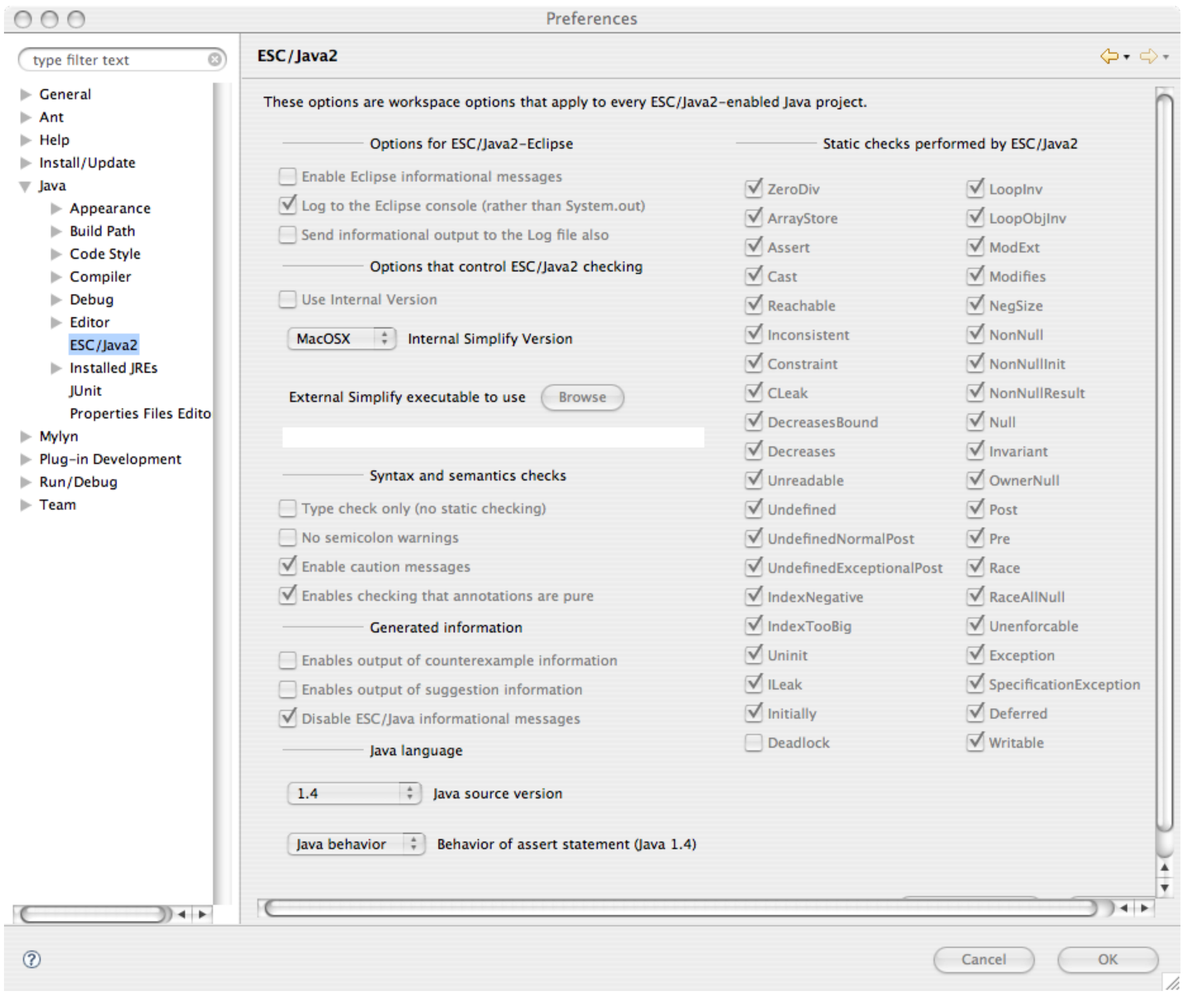

Eclipse File Edit Source Refactor Navigate Search Project JML Run Window Help
ESC/Java2
Clear ESC/Java2 Markers
Go to associated declaration
Enable auto ESC/Java2 checking
Disable auto ESC/Java2 checking
Setup
Package Explorer
Hierarchy
ESCJava2 [src/escjava/trunk/ESCTools]
JML_FM06
JML4Eclipse [jml4/trunk/JML4Eclipse]

Figure 3.6: Enabling the ESC/Java Nature.



ESC/Java2

These options are workspace options that apply to every ESC/Java2-enabled Java project.

Options for ESC/Java2-Eclipse

- ☐ Enable Eclipse informational messages
- ✓ Log to the Eclipse console (rather than System.out)
- ☐ Send informational output to the Log file also

Options that control ESC/Java2 checking

- ☐ Use Internal Version

MacOSX — Internal Simplify Version

External Simplify executable to use

Syntax and semantics checks

- ☐ Type check only (no static checking)
- ☐ No semicolon warnings
- ✓ Enable caution messages
- ✓ Enables checking that annotations are pure

Generated information

- ☐ Enables output of counterexample information
- ☐ Enables output of suggestion information
- ✓ Disable ESC/Java informational messages

Java language

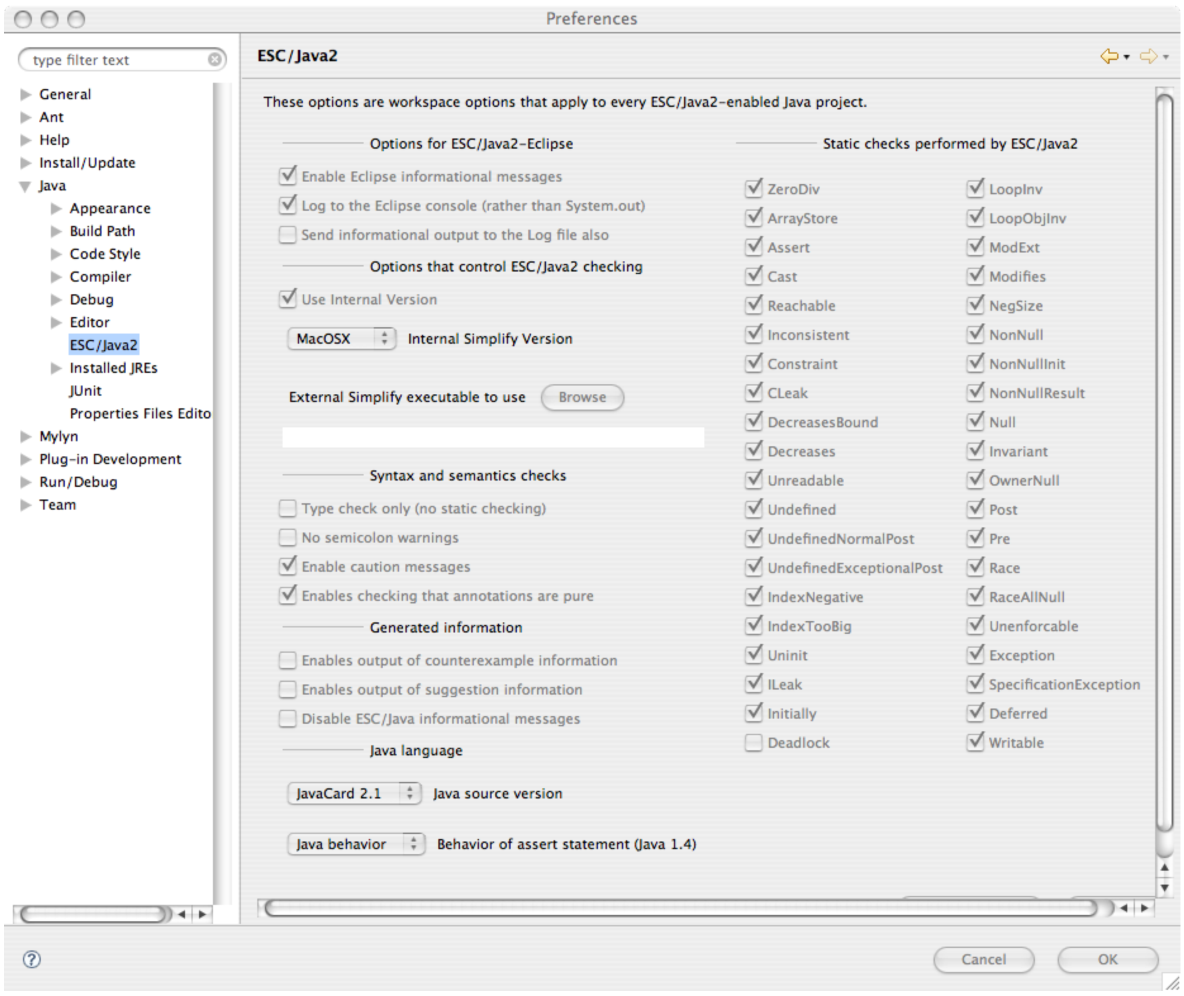
1.4 — Java source version

Java behavior — Behavior of assert statement (Java 1.4)

Static checks performed by ESC/Java2

✓ ZeroDiv	✓ LoopInv
✓ ArrayStore	✓ LoopObjInv
✓ Assert	✓ ModExt
✓ Cast	✓ Modifies
✓ Reachable	✓ NegSize
✓ Inconsistent	✓ NonNull
✓ Constraint	✓ NonNullInit
✓ CLeak	✓ NonNullResult
✓ DecreasesBound	✓ Null
✓ Decreases	✓ Invariant
✓ Unreadable	✓ OwnerNull
✓ Undefined	✓ Post
✓ UndefinedNormalPost	✓ Pre
✓ UndefinedExceptionalPost	✓ Race
✓ IndexNegative	✓ RaceAllNull
✓ IndexTooBig	✓ Unenforcable
✓ Uninit	✓ Exception
✓ ILeak	✓ SpecificationException
✓ Initially	✓ Deferred
☐ Deadlock	✓ Writable

Figure 3.7: Configuring ESC/Java2.



Preferences

type filter text

- General
- Ant
- Help
- Install/Update
- Java
  - Appearance
  - Build Path
  - Code Style
  - Compiler
  - Debug
  - Editor
  - ESC/Java2
  - Installed JREs
  - JUnit
  - Properties Files Edito
- Mylyn
- Plug-in Development
- Run/Debug
- Team

ESC/Java2

These options are workspace options that apply to every ESC/Java2-enabled Java project.

Options for ESC/Java2-Eclipse

- ✓ Enable Eclipse informational messages
- ✓ Log to the Eclipse console (rather than System.out)
- ☐ Send informational output to the Log file also

Options that control ESC/Java2 checking

- ✓ Use Internal Version
- MacOSX — Internal Simplify Version
- External Simplify executable to use — Browse

Syntax and semantics checks

- ☐ Type check only (no static checking)
- ☐ No semicolon warnings
- ✓ Enable caution messages
- ✓ Enables checking that annotations are pure

Generated information

- ☐ Enables output of counterexample information
- ☐ Enables output of suggestion information
- ☐ Disable ESC/Java informational messages

Java language

- JavaCard 2.1 — Java source version
- Java behavior — Behavior of assert statement (Java 1.4)

Static checks performed by ESC/Java2

✓ ZeroDiv	✓ LoopInv
✓ ArrayStore	✓ LoopObjInv
✓ Assert	✓ ModExt
✓ Cast	✓ Modifies
✓ Reachable	✓ NegSize
✓ Inconsistent	✓ NonNull
✓ Constraint	✓ NonNullInit
✓ CLeak	✓ NonNullResult
✓ DecreasesBound	✓ Null
✓ Decreases	✓ Invariant
✓ Unreadable	✓ OwnerNull
✓ Undefined	✓ Post
✓ UndefinedNormalPost	✓ Pre
✓ UndefinedExceptionalPost	✓ Race
✓ IndexNegative	✓ RaceAllNull
✓ IndexTooBig	✓ Unenforcable
✓ Uninit	✓ Exception
✓ ILeak	✓ SpecificationException
✓ Initially	✓ Deferred
☐ Deadlock	✓ Writable

Cancel OK

Figure 3.8: Configuring ESC/Java2 for use with JavaCard.

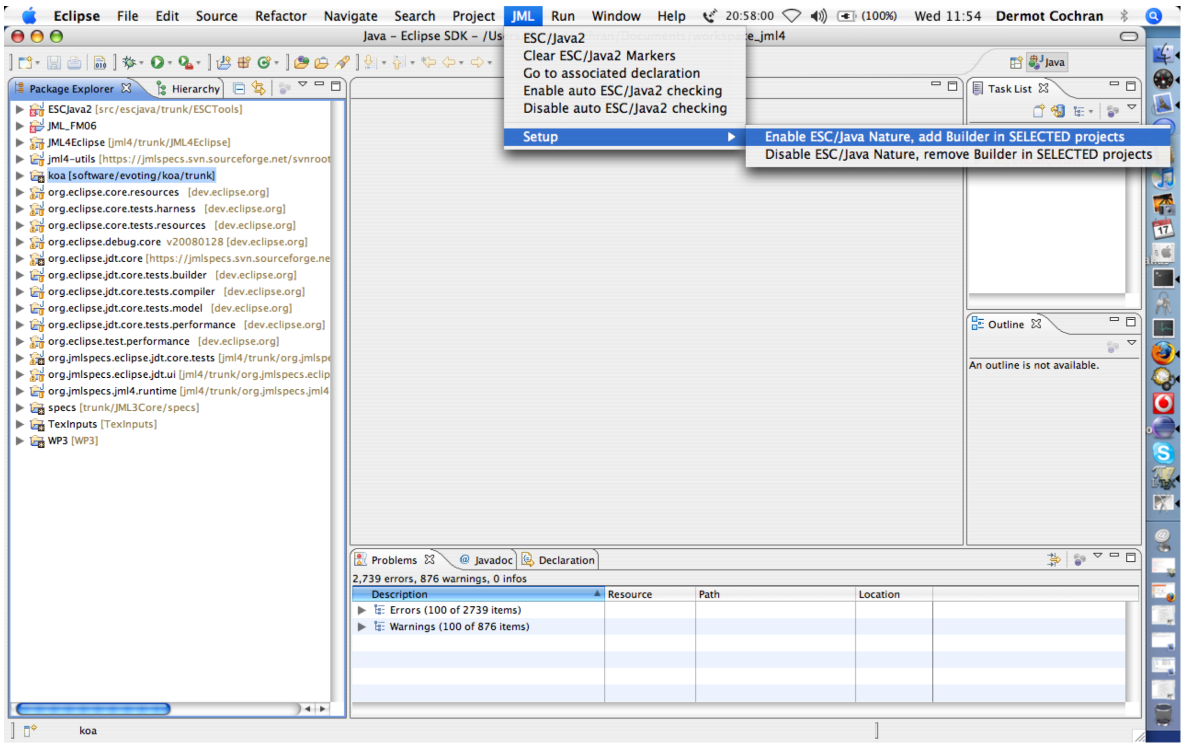

JML
ESC/Java2
Clear ESC/Java2 Markers
Go to associated declaration
Enable auto ESC/Java2 checking
Disable auto ESC/Java2 checking
Setup
Enable ESC/Java Nature, add Builder in SELECTED projects
Disable ESC/Java Nature, remove Builder in SELECTED projects
Package Explorer
Hierarchy
Task List
Outline
An outline is not available.
Problems
Javadoc
Declaration
2,739 errors, 876 warnings, 0 infos
Description
Resource
Path
Location
Errors (100 of 2739 items)
Warnings (100 of 876 items)

Figure 3.9: Enabling automatic execution of ESC/Java2.

Executing MOBIUS and non-MOBIUS Static Checkers

All static checkers can be executed manually with a right mouse button action and/or a menu operation, typically at multiple levels of granularity (e.g., check a method, a class, a file, a package, etc.). Some static checkers like CheckStyle, PMD, FindBugs, and ESC/Java2 can also be configured as autobuilders, which are triggered each time a resource changes.

Generating a Verification Condition

As discussed in several sections earlier, verification conditions are generated by three different subsystems shipped with the Mobius PVE at the moment.

VC generation from ESC/Java2 is built-in and VCs are only retained if special switches, currently unavailable via the Eclipse ESC/Java2 plugin, are used.

VC generation with the MOBIUS DirectVCGen can be controlled using a graphical interface inside of Eclipse Figure 3.11. The FreeBoogie VCGen is only controlled via command-line tools at this time.

Executing the MOBIUS Prover Editor

The MOBIUS Prover Editor is made out of two panels Figure 3.12. It is automatically loaded and initialized anytime a Coq theory file or proof script file is loaded within the Mobius PVE. No special configuration is necessary. If the prover needs to be reset, a small reinitialize button is available in the toolbar.

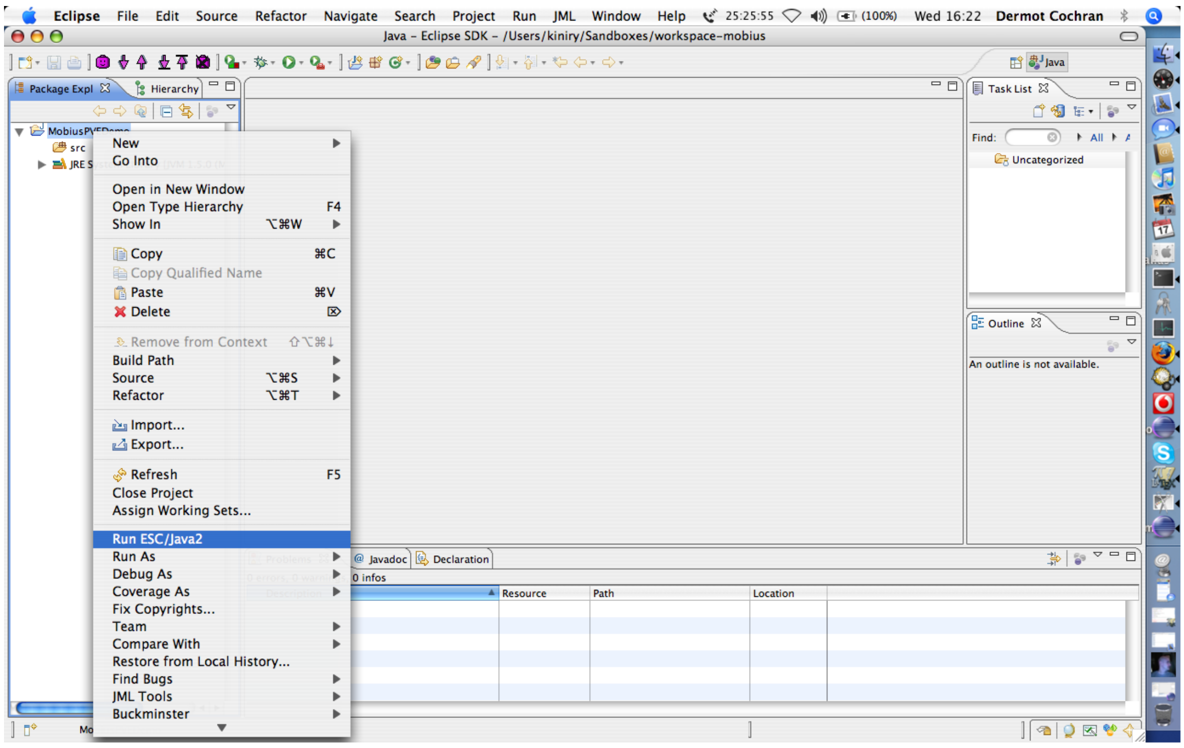
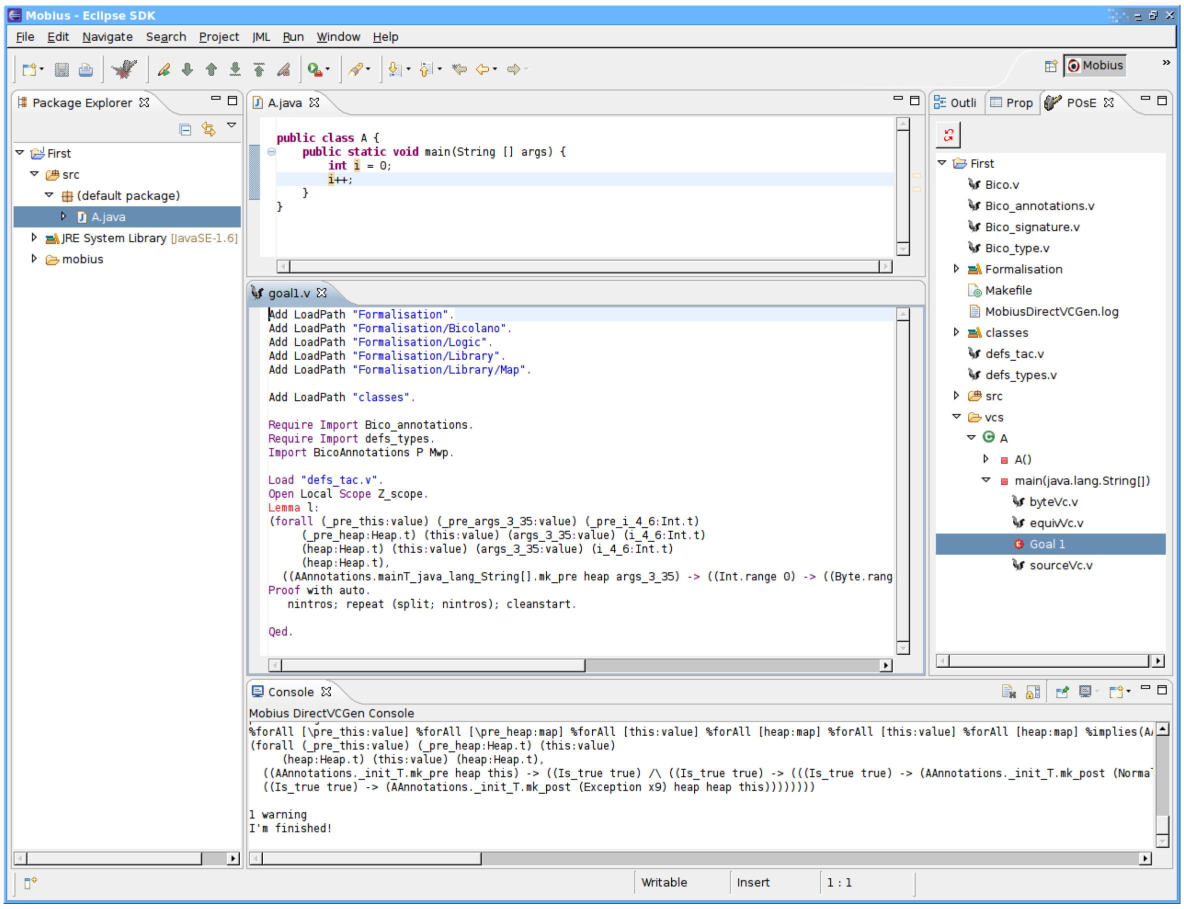


Figure 3.10: Manually Executing ESC/Java2.

Viewing and Manipulating BML-annotated Java Bytecode

Finally, viewing and editing Java bytecode with Umbra Figure 3.13 is accomplished by simply opening a bytecode file or pushing an Umbra button when in a Java source code editor. Except the button to open the bytecode view the environment has also a button to compile JML specifications into a bytecode file in the form of BML annotations.

While viewing bytecode, a variety of buttons are available that permit one to refresh the view, jump to source corresponding to the currently selected bytecode fragment. This completes our brief summary of installation, configuration, and use of the Mobius PVE.

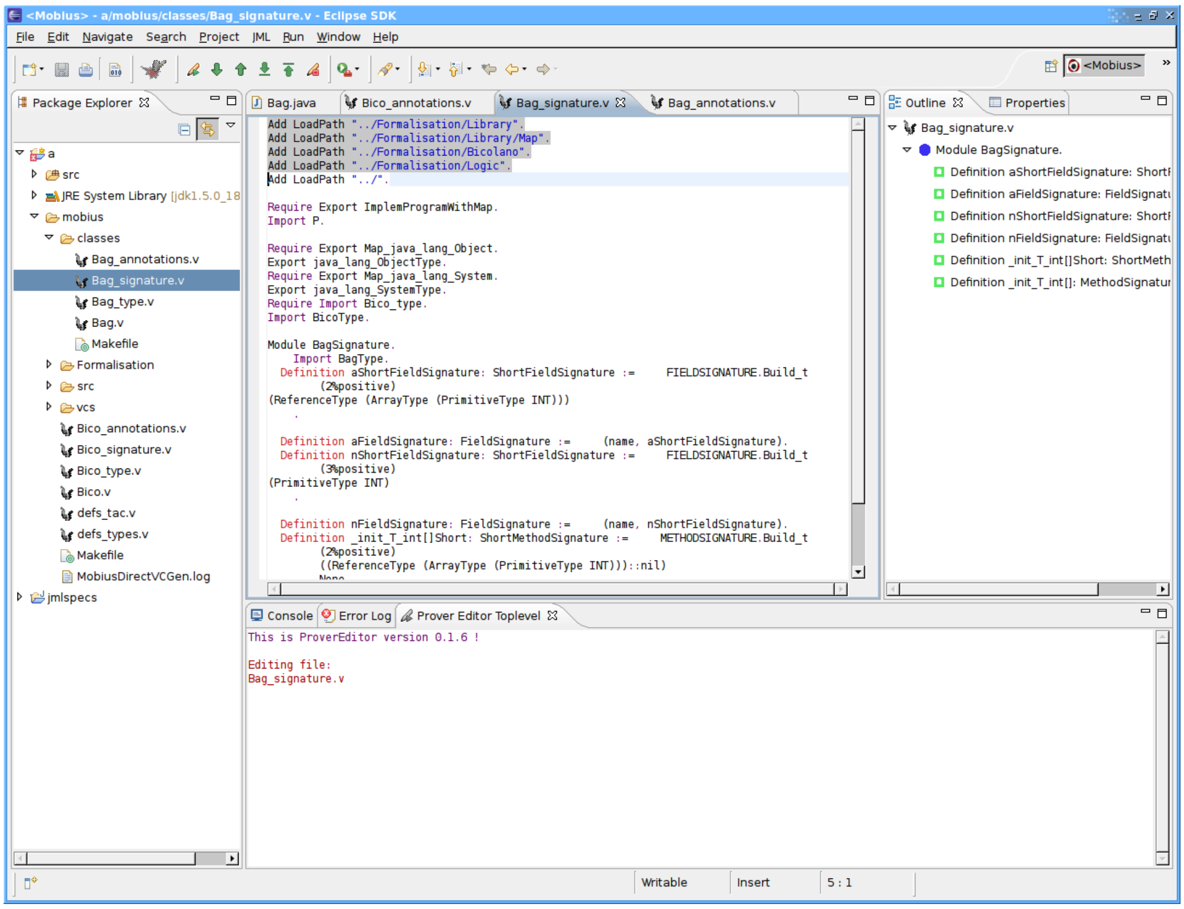


```
public class A {
    public static void main(String [] args) {
        int i = 0;
        i++;
    }
}
```

```
Add LoadPath "Formalisation".
Add LoadPath "Formalisation/Bicolano".
Add LoadPath "Formalisation/Logic".
Add LoadPath "Formalisation/Library".
Add LoadPath "Formalisation/Library/Map".

Add LoadPath "classes".

Require Import Bico_annotations.
Require Import defs_types.
Import BicoAnnotations P Mwp.

Load "defs_tac.v".
Open Local Scope Z_scope.
Lemma l:
(forall (_pre_this:value) (_pre_args_3_35:value) (_pre_i_4_6:Int.t)
     (_pre_heap:Heap.t) (this:value) (args_3_35:value) (i_4_6:Int.t)
     (heap:Heap.t) (this:value) (args_3_35:value) (i_4_6:Int.t)
     (heap:Heap.t),
  ((AAnnotations.mainT_java_lang_String[].mk_pre heap args_3_35) -> ((Int.range 0) -> ((Byte.rang
Proof with auto.
   nintros; repeat (split; nintros); cleanstart.

Qed.
```

```
%forAll [\pre_this:value] %forAll [\pre_heap:map] %forAll [this:value] %forAll [heap:map] %forAll [this:value] %forAll [heap:map] %implies(A
(forall (_pre_this:value) (_pre_heap:Heap.t) (this:value)
     (heap:Heap.t) (this:value) (heap:Heap.t),
  ((AAnnotations._init_T.mk_pre heap this) -> ((Is_true true) /\ ((Is_true true) -> (((Is_true true) -> (AAnnotations._init_T.mk_post (Norma
  ((Is_true true) -> (AAnnotations._init_T.mk_post (Exception x9) heap heap this)))))))

1 warning
I'm finished!
```

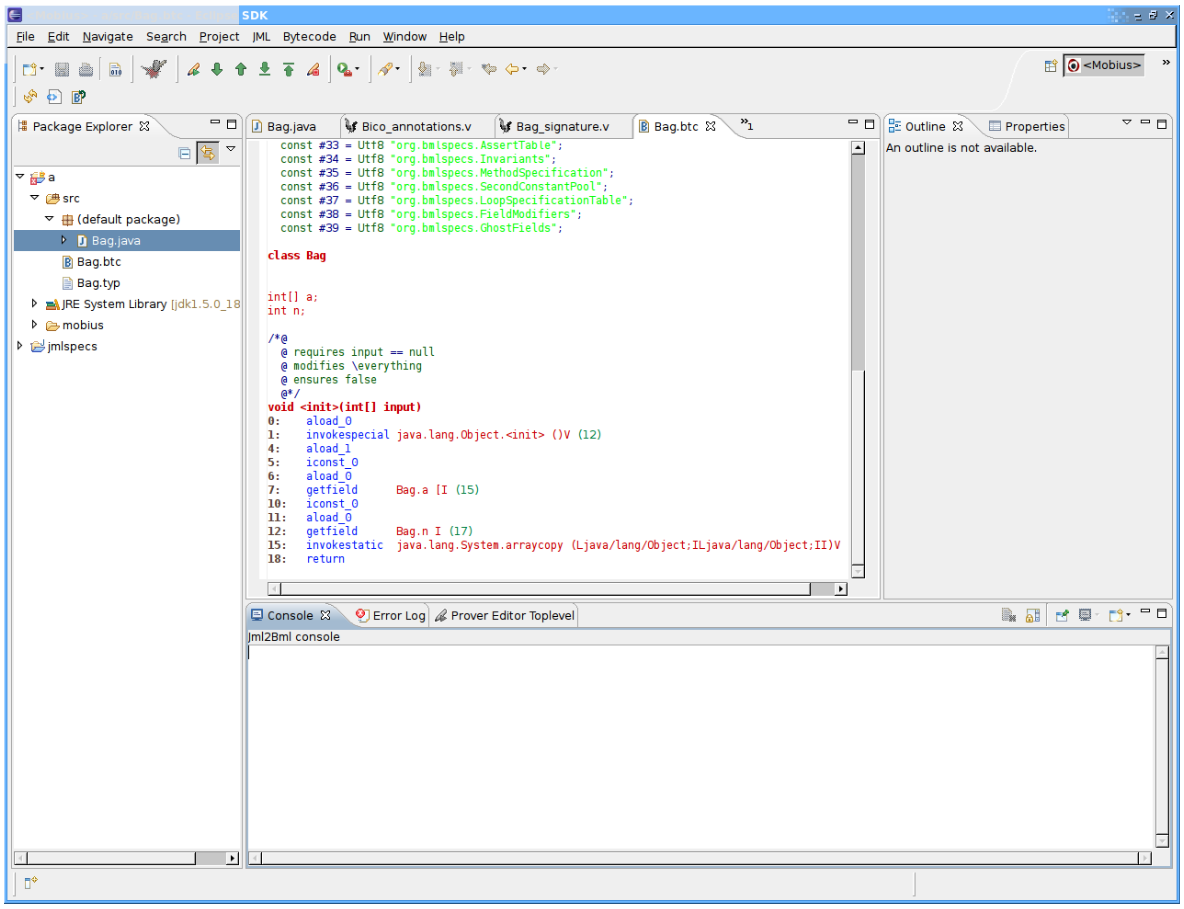
Figure 3.11: Generating and inspecting verification conditions with the DirectVCGen.



```
Add LoadPath "../Formalisation/Library".
Add LoadPath "../Formalisation/Library/Map".
Add LoadPath "../Formalisation/Bicolano".
Add LoadPath "../Formalisation/Logic".
Add LoadPath "../".

Require Export ImplemProgramWithMap.
Import P.

Require Export Map_java_lang_Object.
Export java_lang_ObjectType.
Require Export Map_java_lang_System.
Export java_lang_SystemType.
Require Import Bico_type.
Import BicoType.

Module BagSignature.
    Import BagType.
  Definition aShortFieldSignature: ShortFieldSignature :=     FIELDSIGNATURE.Build_t
        (2%positive)
(ReferenceType (ArrayType (PrimitiveType INT)))
    .

  Definition aFieldSignature: FieldSignature :=     (name, aShortFieldSignature).
  Definition nShortFieldSignature: ShortFieldSignature :=     FIELDSIGNATURE.Build_t
        (3%positive)
(PrimitiveType INT)
    .

  Definition nFieldSignature: FieldSignature :=     (name, nShortFieldSignature).
  Definition _init_T_int[]Short: ShortMethodSignature :=     METHODSIGNATURE.Build_t
        (2%positive)
        ((ReferenceType (ArrayType (PrimitiveType INT)))::nil)
```

```
This is ProverEditor version 0.1.6 !

Editing file:
Bag_signature.v
```

Figure 3.12: Proving in Coq with ProverEditor.



```
  const #33 = Utf8 "org.bmlspecs.AssertTable";
  const #34 = Utf8 "org.bmlspecs.Invariants";
  const #35 = Utf8 "org.bmlspecs.MethodSpecification";
  const #36 = Utf8 "org.bmlspecs.SecondConstantPool";
  const #37 = Utf8 "org.bmlspecs.LoopSpecificationTable";
  const #38 = Utf8 "org.bmlspecs.FieldModifiers";
  const #39 = Utf8 "org.bmlspecs.GhostFields";

class Bag


int[] a;
int n;

/*@
  @ requires input == null
  @ modifies \everything
  @ ensures false
  @*/
void <init>(int[] input)
0:    aload_0
1:    invokespecial java.lang.Object.<init> ()V (12)
4:    aload_1
5:    iconst_0
6:    aload_0
7:    getfield      Bag.a [I (15)
10:   iconst_0
11:   aload_0
12:   getfield      Bag.n I (17)
15:   invokestatic  java.lang.System.arraycopy (Ljava/lang/Object;ILjava/lang/Object;II)V
18:   return
```

Figure 3.13: Viewing and annotating Java bytecode.

Chapter 4

The MOBIUS Software Development Process

Software developed within the MOBIUS project takes place using a variety of tools and processes. As the Mobius PVE has been under development during the first two years of the project, MOBIUS developers are now able to use the prototype Mobius PVE to verify existing and ongoing work. Likewise, software development processes change to accommodate new verification tools and related technologies as they become available. Most developers have used, to date, tools and processes which are related to the intent of the Mobius PVE, e.g., older versions of the JML tools or standard open source static checkers.

The core elements of any medium-to-large scale software development effort, a coding standard, a collaborative development environment, and a version control repository, were put in place very early in MOBIUS development. This chapter briefly summarizes these resources and the current and intended use of each.

4.1 The MOBIUS Coding Standards

The MOBIUS coding standard is derived from the UCD coding standard, originally developed by Joe Kiniry while at Caltech in 1996, then updated while he worked at Radboud University Nijmegen from 2002–2004. This standard has influenced the programming standards of many other research groups and organizations, including Sun Microsystems.

This coding standard stipulates the syntactic style of Java source code including identifier naming, code layout, documentation style and completeness, etc. Additionally, guidelines on appropriate values of a variety of metrics measured by subsystems like those described in subsection 2.2.6, are provided.

The base MOBIUS coding standard is codified in as a CheckStyle style and an Eclipse code formatter configuration. Variant coding styles, specific to various projects, groups, development teams, and sites, are derived from this base standard. They are also codified using CheckStyle and Eclipse. Example "moderate" and "rigorous" standards are provided with the Mobius PVE, as discussed in subsection 2.2.6.

While not all source for all subsystems has been checked against a specific coding standard as of yet, the majority of MOBIUS-specific Mobius PVE code has been checked against one of three different standards. As Mobius PVE development continues, and the MOBIUS development process evolves and becomes more codified within the Mobius PVE, new and existing subsystems are checked against new and existing styles.

4.2 The MOBIUS Collaborative Development Environments

Several web-based Collaborative Development Environments (CDEs) have been used for Mobius PVE development thus far.

Tools developed prior to the MOBIUS project's genesis were hosted at open CDEs sites like SourceForge and private CDEs like the SoS Bugzilla and the UCD GForge.

The initial CDEs used for MOBIUS coordination and some development effort were GForges hosted by INRIA (located at http://mobius.inria.fr) and UCD (located at http://sort.ucd.ie/).

To better coordinate development on the Mobius PVE, a single CDE has been configured at UCD, as it leads the main task relating to Mobius PVE development (Task 3.6). A Trac server, securely hosted at UCD, now hosts nearly all of Mobius PVE development. Existing collaborative artifacts like bug and feature request tickets, patches, repository contents, and wiki pages have been migrated from the various older CDEs mentioned above into this Trac server.

4.2.1 The MOBIUS Trac

The MOBIUS Trac is an integrated project management system, which includes a wiki for technical documentation, a Subversion server for source code (and other artifacts) control, and a ticket management system for defect reports and feature requests. It is integrated with Eclipse using the Mylyn plugin, as described in subsection 2.2.1, which is distributed as part of the Mobius PVE.

The MOBIUS Trac is accessed via the URL https://mobius.ucd.ie/.

4.3 The MOBIUS Version Control Repository

Like CDEs, a variety of version control repositories have been used over the past several years as well. Consolidation of these repositories is ongoing. The MOBIUS Subversion repository is hosted as part of the Trac discussed in the last section.

Consolidation and reorganization of the Subversion repository is still ongoing. For example, the jars and and libs directories are being combined and reorganized.

A Mobius PVE development mailing list called mobius-tools also exists. It is currently hosted at INRIA but will be moved to UCD as well for better integration with the MOBIUS Trac server.

This concludes this summary of the MOBIUS software development process and related servers and tools.

Chapter 5

Epsilon Release Provisional Manifest

The "Epsilon" release of the Mobius PVE is based upon the Eclipse Galileo (3.5.x) IDE for Java Developers.

5.1 Features and Plugins Included

The following MOBIUS plugins and features are included in the "Epsilon" release. The versions of each plugins are the minimum versions of each of them that will be found in the "Epsilon" release.

- Java Modeling Language (JML) feature 1.0.2:
  - JML2 Eclipse plugin 1.0.4
  - JML folding plugin 1.0.7
- Extended Static Checking (ESC) feature 2.0.21:
  - ESC/Java2 plugin 2.0.9
  - ESC/Java2 UI plugin 2.0.14
  - Javafe plugin 2.0.9
  - FreeBoogie plugin 1.0.0
  - RCC plugin 1.0.0
- Automated Theorem Provers (ATP) feature 0.2.0:
  - Simplify plugin 1.5.6
  - CVC3
  - Microsoft Research's Z3 theorem prover 2.0
  - Yices plugin 1.0.21
  - CVC3 plugin
- ProverEditor feature 1.0.2:
  - Prover Editor 0.9.7

  - Coq Editor 0.9.4
  - Coq Sugar 0.9.4

- Bytecode Modeling Language (BML) feature 0.0.14:
  - JavaVerifier 0.0.1
  - BML2BPL 0.0.1
  - BmlLib 1.1.9
  - JML2BML 1.0.1
  - Umbra 1.0.9
  - CCT plugin 0.0.1
  - BMLVCGen plugin 0.0.1

- DirectVCGen feature 1.0.7:
  - Bicolano 1.0.1
  - Bico 1.0.1
  - DirectVCGen 1.0.0
  - DirectVCGenUI 1.0.3

and the following third-party plugins and features:

- BCEL plugin 5.2.0
- Checkstyle feature 4.4.2
- Subclipse feature 1.6.2
- Buckminster 0.2.0
- EclEmma Java Code Coverage feature 1.3.0
- FindBugs feature 1.2.1
- Metrics feature 1.3.6
- Mylyn feature 2.1.0
- PMD feature 3.2.4

5.2 Sample Workspace

The sample workspace contains the following projects:

- ESCJava2 plugin demo
- ETAPS Tutorial demo
- Examples

5.3 Platforms

The “gamma,” “delta,” and “epsilon” releases have been tested on the following platforms:

- Mac OS X 10.4 Tiger
- Mac OS X 10.5 Leopard
- Linux - Ubuntu 9
- Windows XP
- Windows Vista

Chapter 6

Annotation generation

Verifying a program typically requires many program annotations. To make this feasible in practice, some support for automatically generation annotations is vital. Different ways to alleviate the annotation burden have been pursued in Mobius, incl.

- the inference of basic specifications, for instance non-nullness and requirements stemming from array bounds [CFJ+06, BBC+07a].
- dedicated inference algorithms, in particular for decreases clauses [FJ09] and immutability [HP09a, HP09b].
- the generation of annotations from higher level specifications in the form of state machine of various kinds [PS07, HT09, Hui09], more in particular so-called midlet navigation graphs [Cré06];
- the extraction of such higher level specifications from code [Cré08];

These efforts are described in more detail below.

6.1 Inferring and Propagating Annotations

Error messages produced by a program analysis tool, which indicate failed verification attempts, can be used to automate the generation and propagation of program annotations in code. This approach is pursued in the CANAPA tool [CFJ+06]: CANAPA uses the error messages of ESC/Java2 to infer and propagate non-null annotations. CANAPA comes with an Eclipse plug-in.

In an attempt to further reduce the burden of annotation writing further, we have implemented an algorithm dedicated to the generation of "obvious" minimal preconditions to avoid not only null-pointer, but also array-out-of-bounds exceptions (see [BBC+07a]). This algorithm re-uses the implementation of the wp-calculus. A possibility to improve the algorithm is to apply a simple analysis to identify possible class invariants.

It is important to be able to determine the effectiveness of a particular method of specification generator. In order to be able to do that a CodeStatistics tool has been developed which allows to gather statistics on code fragments that enjoy particular property (e.g. for which an annotation that guarantees certain property can be generated). This has been used in a case study to generate JML decreases clauses that guarantee termination of for loops [FJ09].

For the type system for enforcing immutability of objects developed in Task 3.3 [HPSS07] we developed an system to infer the necessary annotations, reported in [HP09b]. Our aim was to only require annotations for immutability that could be handled with the new JSR 308 annotation syntax. The type system for immutability was considerably generalised and streamlined in the process, as reported in [HP09a], but unfortunately this meant that implementation of the type system (as JSR 308 pluggable checker) and the implementation of the inference algorithm have not been finished yet.

6.2 State Diagrams and Navigation Graphs

An alternative way to alleviate the burden of annotating code is to provide some higher-level specification formalism, which makes developing specifications less work. These high-level specifications are then automatically translated into to more basic program annotations.

The earlier work on security requirements reported in Deliverable 1.2 [mC06] suggested that so-called midlet navigation graphs [Cré06] could be such a higher-level specification formalism. Such graphs are essentially a form of security automata. The NAVMID tool developed at France Télécom provides support in drawing midlet navigation graphs. The existing industry standard for testing MIDP application, the Unified Testing Criteria (UTC) [Ini06] already prescribes the use of a similar, albeit completely informal, notion of 'flow diagram.' Unfortunately, the notion of flow diagrams is completely informal [1].

6.2.1 From Java Code to Automata

A formalisation of the notion of navigation graph, which had earlier been sketched in Deliverable D5.1 [mC07c], is given in [Cré08]. The paper also describes an algorithm for navigation graph from Java bytecode. A prototype of this algorithm has been implemented on top of the MATOS program analyser [CA05] at France Télécom. This work is described in more detail in Deliverable D5.2.

6.2.2 From Automata to JML

Security automata are a convenient way to describe security policies. Traditionally they are used to monitor an application at runtime, and to interrupt it as soon as it violates the security policy. However, in our setting of PCC, we would like to use security automata as JML specifications, and verify adherence to them statically. The JML annotations effectively inline the monitor in the application.

The translation of security automata into program annotations is not trivial, and there is the danger that the semantics of the annotations is not equivalent to the effect of runtime monitoring. Therefore we formalised this translation.

The translation from security automata to program annotations is defined in several steps:

1. complete the security automata, adding a special trap state that should not be reached (whereas partial automata, describing only the allowed transitions are sufficient for monitoring);

[1]UTC 2.1 requires the flow diagram to be supplied as JPG or GIF.

2. generate special set-annotations at the level of method specifications, capturing the behavior of the security automata, together with an invariant stating that the error state should not be reached; and

3. inline the annotations into the method body.

We have developed a generic program semantics that is instantiated for monitoring and run-time annotation checking. We prove that every translation step preserves the program behavior. In particular this means that monitoring will not find a security violation iff run-time annotation checking will not find any annotation violation (provided that earlier existing annotations are not violated either).

The translation and its correctness proof have been presented in [HT09]. The correctness proofs revealed several subtleties that have to be considered in the algorithm, and necessary assumptions about program behavior. It also revealed a subtle interplay between the semantics of finally clauses and the behaviour of a run-time checker [Hui09]: abrupt termination of a finally clause can discard a JML annotation error.

Using an annotation propagation algorithm (as in [PBB^{+}04]) the generated annotations give rise to pre- and postcondition annotations in the whole application, that can be verified statically.

In the simplest form of security automata, each edge corresponds to some unique event, typically some method call, but for convenient specifications one can allow richer languages for decorating edges. In [PR08] we define suitable notions of refinement for MIDP navigation graphs, and propose a translation of these to JML annotations.

We have also adapted AutoJML, a tool we developed earlier in Nijmegen [HO03] to generate JML annotations from automata-like specifications, to cope with richer forms of automata. In [PS07] this has been used to generate JML annotations to verify that the MIDP-SSH implementation is correct with respect to a formal specification of the SSH protocol. Here our formal specification probably goes into much more detail than one would typically give in a security automaton.

Work on case studies in checking midlets against navigation graphs is reported in Deliverable 5.2 [mC09] and in [MP09].

Chapter 7

Publications Relating to WP3.6 and WP3.7

The following publications by MOBIUS partners relate directly to subsystems and functionality of the Mobius PVE, case studies performed with the Mobius PVE, or work on annotation generation.

- [BBC$^+$07a] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: A tool for validation of security and behaviour of Java applications. In Formal Methods for Components and Objects: Revised Lectures from the 5th International Symposium FMCO 2006, number 4709 in Lecture Notes in Computer Science, pages 152174. Springer-Verlag, 2007.
- [CHK05] P. Chalin, C. Hurlin, and J. R. Kiniry. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In Verified Software: Tools, Technologies, and Experiences (VSTTE), Zurich, Switzerland, 2005.
- [JC08] J. Charles and J. Kiniry. A lightweight theorem prover interface for eclipse. In User Interfaces for Theorem Proving, 2008.
- [CHS$^+$08] J. Chrzszcz, M. Huisman, A. Schubert, J. Kiniry, M. Pavlova, and E. Poll. BML Reference Manual, December 2008. In Progress. Available from http://bml.mimuw.edu.pl.
- [CHS09] J. Chrzszcz, M. Huisman, and A. Schubert. BML and related tools. In Formal Methods for Components and Objects, number 5751 in Lecture Notes in Computer Science, pages 278–297. Springer-Verlag, 2009.
- [CFJ$^+$06] M. Cielecki, J. Fulara, K. Jakubczyk, . Jancewicz, J. Chrzszcz, A. Schubert, and . Kamiski. Propagation of JML non-null annotations in Java programs. In Principles and Practices of Programming in Java, 2006.
- [CS08] P. Czarnik and A. Schubert. Extending operational semantics of the Java bytecode. In Trustworthy Global Computing, volume 4912 in Lecture Notes in Computer Science, pages 57–72. Springer-Verlag, 2007.
- [Cré08] P. Crégut. Extracting control from data: User interfaces of MIDP applications. In Trustworthy Global Computing: Revised Selected Papers from the Third Symposium TGC 2007, number 4912 in Lecture Notes in Computer Science, pages 4156. Springer-Verlag, 2008.

- [FJ09] J. Fulara and K. Jakubczyk. Practically applicable formal methods. In http://www.mimuw.edu.pl/˜fulara/CodeStatistics/loopconditions.pdf, June 2009. Unpublished manuscript.
- [GM07] R. Grigore and M. Moskal. Edit and verify. In Workshop on First-Order Theorem Proving, September 2007.
- [GCFK09] R. Grigore, J. Charles, F. Fairmichael, and J.R. Kiniry. Strongest Postcondition of Unstructured Programs. In Workshop on Formal Techniques for Java Programs, 2009.
- [HP09a] C. Haack and E. Poll. Type-based object immutability with flexible initialization. In ECOOP 2009, volume 5653 of Lecture Notes in Computer Science, pages 520 545. Springer, 2009.
- [HP09b] C. Haack and E. Poll. Type-based object immutability with flexible initialization (long version). Technical Report ICISR09001, Radboud University Nijmegen, 2009.
- [Hui09] M. Huisman. On the interplay between the semantics of java's finally clauses and the jml run-time checker. In A. Banerjee, editor, Workshop on Formal Techniques for Java Programs. ACM, 2009.
- [HT09] M. Huisman and A. Tamalet. A formal connection between security automata and JML annotations. In Fundamental Approaches to Software Engineering, volume 5503 of Lecture Notes in Computer Science, pages 340354. Springer-Verlag, 2009.
- [Jan07] M. Janota. Assertion-based loop invariant generation. In Workshop on Invariant Generation, Hagenberg, Austria, 2007. Workshop at CALCULEMUS 2007.
- [JGM07] M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In Specification and Verification of Component-Based Systems, a workshop at ESEC/FSE, 2007.
- [JK07] M. Janota and J. Kiniry. Reasoning about feature models in higher-order logic. In Proceedings of the 11th International Conference, SPL, 2007.
- [KMD06] J. R. Kiniry, A. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In Specification and Verification of Component-Based Systems, 2006.
- [Kin07] J. R. Kiniry. Formally counting electronic votes (but still only trusting paper). In IEEE International Conference on Engineering of Complex Computer Systems, Auckland, New Zealand, 2007.
- [KF09] J.R. Kiniry and F. Fairmichael. Ensuring Consistency between Designs, Documentation, Formal Specications, and Implementations. In Component Based Software Engineering (CBSE), 2009.
- [KCT07] J.R. Kiniry, D. Cochran, and P. Tierney. A verification-centric realization of e-voting. In International Workshop on Electronic Voting Technologies, Boston, Massachusetts, 2007. Workshop at USENIX/ACCURATE 2007.

- [KMC$^+$07] J.R. Kiniry, A. Morkan, D. Cochran, F. Fairmichael, P. Chalin, M. Oostdijk, and E. Hubbers. The KOA remote voting system: A summary of work to date. In Trustworthy Global Computing, Lucca, Italy, 2006.
- [KZ08] J.R. Kiniry and D.M. Zimmerman. Secret ninja formal methods. In Formal Methods, 2008.
- [LM07] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In M. Huisman and F. Spoto, editors, Bytecode Semantics, Verification, Analysis and Transformation, Electronic Notes in Theoretical Computer Science, 2007.
- [MK07] M. Moskal, J. opuszaski, and J. Kiniry. E-matching for fun and profit. In The 5th International Workshop on Satisfiability Modulo Theories (SMT), Berlin, Germany, July 2007.
- [PS07] E. Poll and A. Schubert. Verifying an implementation of SSH. In Workshop on Issues in the Theory of Security, pages 164–177. IFIP WG1.7, 2007.
- [SC06] A. Schubert and J. Chrzszcz. ESC/Java2 as a tool to ensure security in the source code of Java applications. In Software Engineering Techniques: Design for Quality, IFIP, Warsaw, 2006. Springer-Verlag.
- [SCB$^+$08] A. Schubert, J. Chrzszcz, T. Batkiewicz, J. Paszek, and W. Ws. Technical aspects of class specification in the byte code of Java language. In To be published in Bytecode'08 proceedings, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

Bibliography

[BBC+07a] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: A tool for validation of security and behaviour of Java applications. In Formal Methods for Components and Objects: Revised Lectures from the 5th International Symposium FMCO 2006, number 4709 in Lecture Notes in Computer Science, pages 152–174. Springer-Verlag, 2007.

[BBC+07b] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. MOBIUS: Mobility, ubiquity, security — objectives and progress report. In TGC 2006 [07].

[BT07] C. Barrett and C. Tinelli. CVC3. In Computer Aided Verification, volume 4590 of Lecture Notes in Computer Science, pages 298–302. Springer, 2007.

[CA05] P. Crégut and C. Alvarado. Improving the security of downloadable Java applications with static analysis. In Bytecode Semantics, Verification, Analysis and Transformation, volume 141 of Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[CFJ+06] M. Cielecki, J. Fulara, K. Jakubczyk, . Jancewicz, J. Chrzszcz, A. Schubert, and . Kamiski. Propagation of JML non-null annotations in Java programs. In Principles and Practices of Programming in Java, 2006.

[CHK05] P. Chalin, C. Hurlin, and J. R. Kiniry. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In Verified Software: Tools, Technologies, and Experiences (VSTTE), 2005.

[CHS+08] J. Chrzaszcz, M. Huisman, A. Schubert, J. Kiniry, M. Pavlova, and E. Poll. BML Reference Manual, December 2008. In Progress. Available from http://bml.mimuw.edu.pl.

[CHS09] J. Chrzszcz, M. Huisman, and A. Schubert. BML and related tools. In Formal Methods for Components and Objects, number 5751 in Lecture Notes in Computer Science, pages 278–297. Springer-Verlag, 2009.

[Con04] The JSR 133 Consortium. The Java memory model causality test cases, 2004. http://www.cs.umd.edu/˜pugh/java/memoryModel/CausalityTestCases.html.

[Cré06] Pierre Crégut. Midlet navigation graphs. Unpublished draft, 2006.

[Cré08] Pierre Crégut. Extracting control from data: User interfaces of MIDP applications. In TGC 2007 [08], pages 41–56.

[CS08] P. Czarnik and A. Schubert. Extending operational semantics of the java bytecode. In TGC 2007 [08], pages 57–72.

[Ecl] Eclipse website. www. See http://www.eclipse.org/.

[07] Trustworthy Global Computing: Revised Selected Papers from the Second Symposium TGC 2006, Lucca, Italy, November 7–9, 2006, number 4661 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[08] Trustworthy Global Computing: Revised Selected Papers from the Third Symposium TGC 2007, Sophia-Antipolis, France, November 5–6, 2007, number 4912 in Lecture Notes in Computer Science. Springer-Verlag, 2008.

[FF00] C. Flanagan and S. N. Freund. Type-based race detection for Java. In Programming Languages Design and Implementation, pages 219–232, New York, NY, USA, 2000. ACM Press.

[FJ09] J. Fulara and K. Jakubczyk. Practically applicable formal methods. In http://www.mimuw.edu.pl/~fulara/CodeStatistics/loopconditions.pdf, June 2009.

[FJS08] J. Fulara, K. Jakubczyk, and A. Schubert. Supplementing java bytecode with specifications. In T. Hruka, L. Madeyski, and M. Ochodek, editors, Software Engineering Techniques in Progress, pages 215–228. Oficyna Wydawnicza Politechniki Wroclawskiej, 2008.

[FL00] C. Flanagan and K.R.M. Leino. Houdini, an annotation assistant for ESC/Java. Technical Report 2000-003, DEC-SRC, December 2000.

[Fur86] G.W. Furnas. Generalized fisheye views. ACM SIGCHI Bulletin, 17(4):16–23, April 1986.

[GCFK09] R. Grigore, J. Charles, F. Fairmichael, and J.R. Kiniry. Strongest Postcondition of Unstructured Programs. In Workshop on Formal Techniques for Java Programs, 2009.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, Third Edition, chapter 17. Sun Microsystems, 2005. http://java.sun.com/docs/books/jls/.

[GM07] R. Grigore and M. Moskal. Edit and verify. In Workshop on First-Order Theorem Proving, September 2007.

[HM05] C. A. R. Hoare and J. Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. A companion paper for VSTTE 2005, held in Züich, Switzerland., July 2005.

[HO03] E. Hubbers and M. Oostdijk. Generating JML specifications from UML state diagrams. In Forum on specification & Design Languages, pages 263–273. University of Frankfurt, 2003. Proceedings appeared as CD-Rom with ISSN 1636-9874.

[HP09a] C. Haack and E. Poll. Type-based object immutability with flexible initialization. In European Conference on Object-Oriented Programming, volume 5653 of Lecture Notes in Computer Science, pages 520–545. Springer, 2009.

[HP09b] C. Haack and E. Poll. Type-based object immutability with flexible initialization (long version). Technical Report ICIS–R09001, Radboud University Nijmegen, 2009.

[HPSS07] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, editor, ESOP'07, volume 4421 of Lecture Notes in Computer Science, pages 347–362. Springer-Verlag, 2007.

[HT09] M. Huisman and A. Tamalet. A formal connection between security automata and JML annotations. In Fundamental Approaches to Software Engineering, volume 5503 of Lecture Notes in Computer Science, pages 340–354. Springer-Verlag, 2009.

[Hui09] M. Huisman. On the interplay between the semantics of java's finally clauses and the jml run-time checker. In A. Banerjee, editor, Workshop on Formal Techniques for Java Programs. ACM, 2009.

[Ini06] Unified Testing Initiative. Unified testing criteria for Java technology-based applications for mobile devices. Technical report, Sun Microsystems, Motorola, Nokia, Siemens, Sony Ericsson, May 2006. Version 2.1.

[Jan07] M. Janota. Assertion-based loop invariant generation. In Workshop on Invariant Generation, 2007. Workshop at CALCULEMUS 2007.

[JC08] J. Kiniry J. Charles. A lightweight theorem prover interface for eclipse. In User Interfaces for Theorem Proving, 2008.

[JGM07] M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In Specification and Verification of Component-Based Systems, Dubrovnik, Croatia, 2007. Workshop at ESEC/FSE 2007.

[JK07] M. Janota and J. Kiniry. Reasoning about feature models in higher-order logic. In Proceedings of the 11th International Conference, SPL, 2007.

[KCT07] J.R. Kiniry, D. Cochran, and P. Tierney. A verification-centric realization of e-voting. In International Workshop on Electronic Voting Technologies, Boston, Massachusetts, 2007. Workshop at USENIX/ACCURATE 2007.

[KF09] J.R. Kiniry and F. Fairmichael. Ensuring consistency between designs, documentation, formal specications, and implementations. In International Symposium on Component Based Software Engineering, 2009.

[Kin98] J. R. Kiniry. IDebug: An advanced debugging framework for Java. Technical Report CaltechCSTR:1998.cs-tr-98-16, California Institute of Technology, 1998.

[Kin04] J.R. Kiniry. Formalizing the user's context to support user interfaces for integrated mathematical environments. Electronic Notes in Theoretical Computer Science, 103, 2004.

[Kin07] J. R. Kiniry. Formally counting electronic votes (but still only trusting paper). In IEEE International Conference on Engineering of Complex Computer Systems, Auckland, New Zealand, 2007.

[KMC$^+$07] Joseph R. Kiniry, Alan E. Morkan, Dermot Cochran, Fintan Fairmichael, Patrice Chalin, Martijn Oostdijk, and Engelbert Hubbers. The KOA remote voting system: A summary of work to date. In TGC 2006 [07], pages 244–262.

[KMD06] J. R. Kiniry, A. Morkan, and B. Denby. Soundness and completeness warnings in ESC/Java2. In Specification and Verification of Component-Based Systems, 2006.

[KO03] J.R. Kiniry and S. Owre. Improving the PVS user interface. In User Interfaces for Theorem Proving, September 2003.

[KZ08] J.R. Kiniry and D.M. Zimmerman. Secret ninja formal methods. In Formal Methods, 2008.

[LM07] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In M. Huisman and F. Spoto, editors, Bytecode Semantics, Verification, Analysis and Transformation, Electronic Notes in Theoretical Computer Science, 2007.

[LPC$^+$05] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML Reference Manual, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

[mC06] MOBIUS Consortium. Deliverable 1.2: Framework-specific and application-specific security requirements, 2006. Available online from http://mobius.inria.fr.

[mC07a] MOBIUS Consortium. Deliverable 4.2: Certificates, 2007. Available online from http://mobius.inria.fr.

[mC07b] MOBIUS Consortium. Deliverable 4.3: Intermediate report on proof-transforming compiler, 2007. Available online from http://mobius.inria.fr.

[mC07c] MOBIUS Consortium. Deliverable 5.1: Selection of case studies, 2007. Available online from http://mobius.inria.fr.

[mC09] MOBIUS Consortium. Deliverable 5.2: Evaluation of type based and logical verification technology, 2009.

[ML06] M. Moskal and J. Lopuszaski. Fast quantifier reasoning with lazy proof explication. http://nemerle.org/ malakith/smt/smt-tr-1.pdf, 2006. Available via http://nemerle.org/.

[MK07] M. Moskal, J. opuszaski, and J. Kiniry. E-matching for fun and profit. In The 5th International Workshop on Satisfiability Modulo Theories (SMT), Berlin, Germany, July 2007.

[MP09] Wojciech Mostowski and Erik Poll. Midlet Navigation Graphs in JML. Technical Report ICIS–R09004, Radboud University Nijmegen, August 2009. Available at https://pms.cs.ru.nl/iris-diglib/src/getContent.php?id=2009-Mostowski-MidletGraphs.

[MR07a] P. Müller and A. Rudich. Formalization of ownership transfer in universe types. Technical Report 556, ETH Zurich, 2007.

[MR07b] P. Müller and A. Rudich. Ownership transfer in universe types. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 2007. To appear.

[PBB$^+$04] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, Smart Card Research and Advanced Application. Kluwer Academic Publishing, 2004.

[Pic06] D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inria.fr/bicolano. Summary appears in [?], 2006.

[PR08] E. Poll and J. N. Ravelo. Generalised navigation graphs, their refinement, and their use for midp applications, 2008. Manuscript. 48 pages.

[PS07] E. Poll and A. Schubert. Verifying an implementation of SSH. In Workshop on Issues in the Theory of Security, pages 164–177. IFIP WG1.7, 2007.

[SC06] A. Schubert and J. Chrzszcz. ESC/Java2 as a tool to ensure security in the source code of Java applications. In Software Engineering Techniques: Design for Quality, IFIP, Warsaw, 2006. Springer-Verlag.

[SCB$^+$08] A. Schubert, J. Chrzszcz, T. Batkiewicz, J. Paszek, and W. Ws. Technical aspects of class specification in the byte code of java language. In To be published in Bytecode'08 proceedings, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[win08] The Wine user guide. Available via http://www.winehq.org/, 2008.

[WN95] K. Waldén and J.-M. Nerson. Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems. Prentice Hall, 1995.