Open Integrated Development and Analysis Environments

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Diplom Wirtschaftsinformatiker Michael Eichberg

geboren in Langen

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr. Oege de Moor
Datum der Einreichung:	20. November 2006
Datum der mündlichen Prüfung:	12. Januar 2007

Erscheinungsjahr 2007

Darmstadt D17

Abstract

Comprehensive tool support is essential to enable developers to cope with the complexity of modern software development projects. Software projects are getting larger and larger, are being developed using different languages, and make use of many third-party libraries as well as frameworks. Hence, tools are required: for software comprehension, for checking that libraries and frameworks are correctly used, and to ensure that the design does not degrade over time.

Though numerous successful tools have already been developed for these tasks [DDL99, HP04, JD03, Fav02, HVdM06, LL05], several issues remain: the tools are usually highly specialized, their extensibility is limited, and an integration between the tools is lacking. Furthermore, IDE integration and in particular an integration with the incremental build process offered by modern IDEs is also often missing. Unfortunately, the direct integration of several code analysis tools with the incremental build process is not possible. When each tool processes the project's resources on its own and also maintains its own source model, the overall memory requirements and analysis time is prohibitive.

To address these issues, this thesis proposes the concept of a *Build Process Integrated Open Static Analysis Platform*. The core functionality of such a platform is to coordinate the execution of static analyses that are encapsulated into modules with well-defined interfaces. These interfaces specify what the analyses require and provide in terms of the data they process. For a tool that is built upon such a platform it is sufficient to specify the data it requires. The platform can then determine the set of analyses and their execution order to satisfy the tool's requirements.

Modeling analyses as modular producer-consumer units facilitates the simultaneous integration of several tools into the incremental build process of modern IDEs. When compared to using several independent tools, the overall memory requirements are reduced, since the source model derived by the executed analyses is shared among all tools built upon the platform. Furthermore, the overall analysis time is also reduced since analyses are executed at

most once, even if the derived information is required by more than one tool. The overall analysis time is further minimized by the parallel execution of those analyses that process different information.

The feasibility of the proposed approach is demonstrated by MAGELLAN. MAGELLAN is an open static analysis platform tightly integrated with the incremental build process of the Eclipse IDE. This integration turns Eclipse into an Integrated Development and Analysis Environment. The set of modules implementing the static analyses is freely extensible and the data model of the database is open. An open data model is crucial to support new analyses that need to store derived information for the use by subsequent analyses.

Besides featuring a fully flexible analysis stack, MAGELLAN also supports the embedding of query engines. Supporting the execution of queries is indispensable for enabling end-users to define application specific analyses. The ability to execute queries is also required to facilitate software comprehension tools. As a proof of concept an XQuery processor and a Prolog system are embedded into MAGELLAN. Both engines are evaluated w.r.t. to using them for the execution of queries along with the incremental build process. The XQuery engine is additionally evaluated in the context of software comprehension tools as a means to enable the end-user to define new ways to navigate through code.

The platform is validated by four tools built on top of it: a software exploration tool, a metrics tool, an optional type system, and a set of lightweight static analyses that check structural properties of source code.

Zusammenfassung

Eine Umfassende Werkzeugunterstützung ist essentiell, um Entwicklern die Beherrschung der Komplexität moderner Softwareentwicklungsprojekte zu ermöglichen. Softwareprojekte werden zunehmend größer, verwenden verschiedene Sprachen und nutzen eine große Anzahl externer Bibliotheken und Frameworks. Vor diesem Hintergrund werden Werkzeuge zur Förderung des Softwareverständnisses benötigt, um zu prüfen, ob Bibliotheken und Frameworks korrekt benutzt werden und um sicherzustellen, dass das Design von Anwendungen während der Entwicklung nicht zerfällt.

Obwohl bereits zahlreiche erfolgreiche Werkzeuge für diese Aufgaben entwickelt wurden [DDL99, HP04, JD03, Fav02, HVdM06, LL05], sind einige Probleme noch ungelöst: die Werkzeuge sind typischer Weise hoch spezialisiert, ihre Erweiterbarkeit ist beschränkt und eine Integration zwischen den Werkzeugen ist nicht vorhanden. Weiterhin ist eine Integration in integrierte Entwicklungsumgebungen (IDEs) und insbesondere eine Einbettung in den inkrementellen Übersetzungsvorgang moderner IDEs meist nicht vorhanden. Eine direkte Integration mehrerer Analysewerkzeuge mit dem inkrementellen Übersetzungsvorgang ist nicht möglich. Wenn jedes Werkzeug den Quelltext des Projekts selbständig verarbeitet und auch ein eigenes Modell der Software wartet, dann sind die Gesamtanforderungen bezüglich Speicherbedarf und Analysezeit zu hoch.

Um diese Probleme zu lösen, wird in dieser Dissertation das Konzept von offenen statischen Analyseplattformen vorgeschlagen, die in den inkrementellen Übersetzungsvorgang eingebunden sind. Die Kernfunktionalität solcher Plattformen ist die Koordination der Ausführung statischer Analysen, welche in Module mit wohl definierten Schnittstellen eingekapselt sind. Die Schnittstellen spezifizieren im Hinblick auf die verarbeiteten Daten, was die Analysen benötigen und zur Verfügung stellen. Für Werkzeuge, die auf diesen Plattformen aufsetzen, ist es ausreichend zu spezifizieren welche Daten benötigt werden. Die Plattform kann dann die Menge der Analysen und ihre Ausführungsreihenfolge bestimmen, um die Anforderungen der Werkzeuge zu erfüllen.

Die Modellierung der Analysen als modulare Produzenten-Konsumenten-Einheiten ermöglicht die gleichzeitige Integration mehrerer Werkzeuge in den inkrementellen Übersetzungsprozess moderner IDEs. Verglichen mit der Nutzung mehrerer unabhängiger Werkzeuge sind die Speicheranforderungen aufgrund der gemeinsamen Nutzung des Softwaremodells reduziert. Das Softwaremodell wird während der Ausführung der Analysen abgeleitet und von allen auf der Plattform aufsetzenden Werkzeugen genutzt. Weiterhin wird die Gesamtanalysezeit dadurch reduziert, dass jede Analyse höchstens einmal ausgeführt wird — insbesondere auch dann, wenn die abgeleitete Information von mehreren Werkzeugen benötigt wird. Darüber hinaus wird die Gesamtanalysezeit minimiert durch die parallele Ausführung von Analysen, die verschiedene Daten verarbeiten.

Die Realisierbarkeit des vorgeschlagenen Ansatzes wird durch Magellan demonstriert. Magellan ist eine offene statische Analyseplattform, die eng in den inkrementellen Übersetzungsprozess der Eclipse IDE integriert ist. Diese Integration verwandelt Eclipse in eine Integrierte Entwicklungs- und Analyseumgebung. Die Menge der Module, die statische Analysen implementieren, ist frei erweiterbar und das Datenmodell ist offen für Ergänzungen. Ein offenes Datenmodell ist unabdingbar, um neue Analysen zu unterstützen, die abgeleitete Information für nachfolgende Analysen zwischenspeichern müssen.

Neben der Unterstützung eines vollständig flexiblen Analysestapels unterstützt Magellan auch das Einbetten von Abfragesprachen. Die Unterstützung der Ausführung von Abfragen ist unverzichtbar, um Endanwendern die Spezifikation von anwendungsspezifischen Analysen zu ermöglichen. Die Fähigkeit Abfragen auszuführen ist auch notwendig, um die Implementierung von Werkzeugen zum Softwareverständnis zu ermöglichen. Die Tragfähigkeit des Konzeptes wird durch die beispielhafte Einbettung eines Prolog Systems und eines XQuery Prozessors in Magellan gezeigt. Beide Ansätze werden im Hinblick auf ihre Eignung zur Ausführung von Abfragen als Teil des inkrementellen Übersetzungsvorgangs bewertet. Der XQuery Prozessor wird zudem bezüglich seiner Verwendung in Softwareverständniswerkzeugen evaluiert — als ein Ansatz um dem Endbenutzer die Definition von neuen Abfragen zur Navigation durch den Code zu ermöglichen.

Die Plattform wird validiert durch vier auf der Plattform aufsetzende Werkzeuge. Dies sind ein Werkzeug zur Exploration von Software, ein Werkzeug zur Berechnung von Metriken, ein optionales Typsystem und eine Menge von leichtgewichtigen statischen Analysen, die strukturelle Eigenschaften des Quellcodes überprüfen.

Contents

List of Figures

List of Tables

List of Listings

Preface

Already when I was a student in my first semesters, I knew that I wanted to pursue a doctorate. But, back then I had no idea what it actually means to do it and how challenging it was going to be. Though I usually prefer to work on my own, I soon realized - after completing my studies and starting my doctorate - that doing serious research completely on one's own is not going to work. Of course, finally wrapping up the thesis is something you have to do on your own, but everything before it, that is, carrying out the necessary research, requires intensive support by many. Besides that, I realized that my studies had not sufficiently prepared me for a career in research — writing a diploma thesis and writing a scientific paper are two completely different things; in particular if english is not your mother tongue. However, the language barrier was only a minor problem. You first have to identify a target area in which you want to do research; you have to become confident in identifying related work and it is also necessary to learn how to judge the quality of related work, how to communicate your ideas, how to structure a paper and how to guide and supervise other students that are supposed to support your work as part of their diploma theses. Due to Prof. Dr. Mira Mezini's support — my supervisor — I was able to quickly overcome these obstacles and after I published my first papers she continued to support me and helped me to further improve my writing and to further clear up my ideas. Thank you very much, without your support this thesis would not have been possible.

Besides Prof. Mezini, I would also like to thank my co-supervisor Prof. Oege de Moor. He provided a very appreciated second view on my thesis. His view of my thesis removed my remaining doubts.

Additionally, I would like to thank everyone who contributed to this thesis, by helping me to write papers or by implementing parts of the research prototypes developed as part of this thesis. I would like to thank (in alphabetical order): Christoph Bockisch, Sinisa Dukanovic, Daniel Germanus, Michael Haupt, Matthias Kahl, Sven Kloppenburg, Karl Klose, Lukas

Mrokon, Klaus Ostermann, Benjamin Rank, Thorsten Schäfer, Tobias Schuh, Mario Vekic and all those I forgot to mention.

Furthermore, I would like to thank Gudrun Jörs, my parents Christel and Werner Eichberg, and my partner Alice Müller. They all supported me in one way or the other. Due to their support, I was able to stay focused on my doctorate and to bring the biggest project of my life (so far) to a successful end.

This work was partially supported by a scholarship of the Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg 492 "Infrastruktur für den elektronischen Markt".

Part I

Introduction

Chapter 1

Overview

> There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success than to take the lead in the introduction of a new order of things.
>
> *Niccolo Machiavelli*

This thesis discusses the design, implementation and evaluation of analysis platforms that facilitate the integration of static analysis based tools with an incremental build process, particularly as offered by modern software development environments. Enabling the simultaneous integration of different code analysis tools promises to further improve (a) the quality of the software and (b) the productivity of developers.

1.1 This Thesis in a Nutshell

Modern software development projects are getting larger and larger, are being developed using different languages, and make use of many third-party libraries as well as frameworks. After the initial deployment the applications need to be maintained over years. Hence, to help developers coping with the complexity of software development, tool support is required: for software comprehension, for checking that libraries and frameworks are correctly used, and to ensure that the design does not degrade over time.

To provide support for the mentioned tasks numerous tools have already been developed [DDL99, HP04, JD03, Fav02, HVdM06, LL05]. But, these tools are usually highly specialized and their extensibility — if at all — is often limited to the particular domain of the tool. For example, Findbugs [HP04] is limited to finding bug patterns and checking structural properties; the detection of errors that, e.g., require whole program data-flow analyses is not supported. Saber [RSS+04b] is delivered with a fixed set of templates that can be parameterized to detect a set of common types of errors, in particular errors related to method call protocols; other types of analyses are not supported. The software visualization tool CodeCrawler [Lan03] provides a metrics based visualization of the structure of an application, but does not support an exploration of the program's control flow.

Hence, developers that want to analyze their projects have to use a multitude of tools. This clearly hinders the adoption of code analysis tools, as also identified in [Vol06]. An approach is lacking that facilitates the development and integration of a wide range of code analysis tools and software comprehension tools.

Besides lacking inter-tool integration, many current tools are still not integrated with modern IDEs. This lack of integration between software engineering tools, and in particular the lack of IDE integration was identified as one of the main reasons why software engineering tools are not widely adopted [FES03].

IDE integration and in particular an integration with the incremental build process offered by modern IDEs promises to further improve the productivity of developers:

- IDE integration reduces the effort of using software comprehension and static code analysis tools; IDE integration enables the effective use of these tools:
  - If tools for finding and preventing errors are integrated into an IDE, the developer can directly navigate to the source code and fix the bug given the error message. It is no longer necessary to manually navigate to the errors identified by the (external) tool.
  - If software comprehension tools are IDE integrated, the developer can directly use the gained knowledge to maintain and evolve the code. It is not necessary to switch between different tools.

    Further, if multiple software comprehension tools are integrated into the IDE, an integration between the different provided views

is possible. Being able to navigate between different views was identified as very useful in a study related to software visualization tools carried out by Bassil and Keller [BK01].

- A tight integration with an IDE's incremental build process enables keeping the source model, which underlies code analysis and comprehension tools, always up-to-date. This makes it is possible to give the developer timely information that helps to asses the effect of the current change [GYF06]:

  - Errors that are immediately reported when they occur are often easier to comprehend and fix. A small change to the type hierarchy of an object-oriented program may cause dozens of cascading errors. Without immediate feedback the developer will continue editing the source code to be confronted with dozens of errors only after the next build or analysis of the project. Tracing the root of the error messages and judging their relevance is time consuming; immediate feedback is much more effective and improves the productivity.

    Further, by executing analyses that complement the compiler's analyses it is possible to detect more errors earlier and, hence, to reduce the development costs [McC93].

  - Software comprehension tools can always immediately be used. It is no longer necessary for developers to wait until the model is updated and the developers' productivity will be increased.

    Examples are tools to visualize and explore the control flow of an application [RSK00, SCHC99] or tools to analyze the mutability[1] of fields [PBKM00].

However, a naïve integration, where each tool implements all functionality — from parsing the code to displaying the errors — on its own, is not feasible. If each tool parses the code and maintains its own source model, the memory and the time required to maintain the models would be too excessive to run several independent tools along with the incremental build process. Furthermore, common functionality is implemented over and over again, e.g., the code that provides the build process integration, the parsers

[1] A field is considered mutable if the field's value is ever updated after the initial initialization.

which derive and maintain the source model during incremental builds, and the code to visualize error reports. Hence, engineering effort is wasted.

To enable the simultaneous build process integration of tools that use code analysis and to reduce the engineering efforts of building such tools this thesis proposes *Open Static Analysis Platforms*. The core idea is to consider code analysis tools as fine-grained modular systems where each analysis is a module with a well-defined interface. The interfaces specify what the analyses require and provide. These specifications are used by the platform to control the interaction between the analyses. Hence, open static analysis platforms are basically coordinators of sets of modularized analyses.

For example, a tool to detect violations of implementation restrictions related to Enterprise JavaBeans [EJB03] could be modularized as follows: one analysis parses the Java source files and extracts the source model, a second analysis derives the call graph by analyzing the source model provided by the first analysis, and a third analysis analyzes the XML deployment descriptors of the components to determine the method's transaction attributes. The information derived by these three analyses is subsequently used by the analyses that actually detect the violations.

Given the analyses' interface specifications the platform will be able to determine a proper schedule for running the analyses. A proper schedule is one that starts each analysis only after the data it requires is available. A schedule for the given example would be to first execute the analysis which parses the Java code. Next the analysis to derive the call graph and the analysis of the XML deployment descriptors can run in parallel. In the last step, the analyses that detect the errors are executed. The schedule is calculated by solving an integer optimization problem. To do so, each analysis is associated with an integer variable that determines the point in time at which to execute the analysis. Further, an analysis' requirements are represented as a set of constraints that ensures that the analysis is only executed when all its requirements are satisfied.

Modeling analyses as modular producer-consumer units facilitates the simultaneous integration of several tools into the incremental build process:

- Analyses are executed at most once, even if the derived information is required by more than one tool.
- The source model derived by executed analyses is shared among all tools.

Hence, the overall processing time as well as the memory requirements are reduced. The engineering effort necessary to develop new tools is also reduced

as it is possible to reuse exactly those analyses that are required for the problem at hand.

Besides minimizing the analysis time and memory requirements when running static analyses, the proposed model also facilitates the integration of engines for querying information about software derived by static analyses. Supporting query engines is indispensable for static analysis platforms. When compared with analyses implemented using procedural or object-oriented programming languages, analyses developed using a declarative language are typically more concise and can be developed in less time, because the developer just has to specify what needs to be computed and not how [Mit03]. An additional benefit of developing analyses using languages such as Prolog or Datalog is that these analyses are often easier to comprehend and maintain. Enabling the evaluation of queries facilitates:

- the development of static code analyses [Cre97, MLL05, HVdM06, Cop06, HCXE02].

- the development of software comprehension tools as demonstrated, e.g., by JQuery [JD03] or the Searchable Bookshelf [SCHC99].

- end-user extensible tools, i.e., tools where additional analyses (queries) can be defined to customize the tool to the specifics of a project. For example, the software exploration tool JQuery [JD03] uses Prolog queries for the exploration of the project, the visualization tool Searchable Bookshelf [SCHC99] uses a tool specific language called GCL, and the static analysis tool CodeQuest [HVdM06] uses Datalog.

In general, tools that make use of query engines employ a two step process. First, the initial source model is derived by analyses that are typically implemented in procedural or object-oriented languages such as C, C++, C# or Java. These *base analyses* store their results in a database. E.g., CodeQuest [HVdM06] uses an SQL database; JQuery [JD03] uses a tool specific internal database. After that, the declarative queries are evaluated against the database. In the proposed approach in this thesis base analyses are modeled as modularized units which specify to maintain the data stored in a database. A tool that wants to make use of a specific query engine then specifies a dependency on the database — to make sure the information stored in the database is maintained — and uses the query engine to evaluate the queries.

1.2 Contributions of this Thesis

The major contributions of this thesis w.r.t. the design, implementation and evaluation of open static code analysis platforms are listed in the following.

- Requirements on open static analysis platforms are identified. Platforms that fulfil the requirements will facilitate the development of software comprehension tools as well as static code analysis tools for finding and preventing errors (Chapter 2).

- An approach to modularizing static analyses is proposed and implemented (Chapter 3). The proposed approach supports analyses that (a) derive new information, (b) update information during incremental builds, and (c) transform information, e.g., one code representation into another code representation.

  Furthermore, the proposed approach minimizes the number of executed analyses to those which directly or indirectly derive information required by the analyses explicitly chosen by the end-user. As part of scheduling the analyses, those analyses are identified that can be executed in parallel.

  Based upon the implementation of the proposed approach it is shown that the development of open static analysis platforms is feasible.

- The scalability of open static analysis platforms w.r.t. the number of additional analyses that can be executed along with an incremental build process is evaluated (Chapter 6).

- The benefits of open static analysis platforms when designing and building software engineering tools on top of them are identified. The benefits are further demonstrated by prototypical implementations of:

  - lightweight static code analyses (Chapter 6)
  - a software exploration tool (Chapter 7)
  - a metrics tool (Chapter 8)
  - a pluggable type systems (Chapter 9)

- The simultaneous integration of different query engines into an open static analysis platform is demonstrated (Chapter 5).

The following contributions, which are related to the development of software engineering tools in general and which are not specific to static analysis platforms, are also worth mentioning.

- It is shown that the use of the declarative query language XQuery facilitates the development of software engineering tools (SEXTANT in Chapter 7 and QSCOPE in Chapter 8).
- It is shown that automatically incrementalized Prolog based analyses are sufficiently fast to be executed along with an incremental build process. This the first application of automatically incrementalized Prolog queries for the implementation of whole program analyses integrated into an IDE (Chapter 9).

In the framework of the research done in this thesis, the following papers have been published:

1. M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. *Automatic Incrementalization of Prolog Based Static Analyses.* In Proceedings of the Ninth International Symposium on Practical Aspects of Declarative Languages (PADL), Volume 4354 of Lecture Notes in Computer Science, pp. 109–123. Springer, 2007.
2. M. Eichberg, M. Mezini, S. Kloppenburg, K. Ostermann, and B. Rank. *Integrating and Scheduling an Open Set of Static Analyses.* In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 113–122. IEEE Computer Society, 2006.
3. T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini. *The Sextant Software Exploration Tool.* IEEE Transactions on Software Engineering, vol. 32 (no. 9), pp. 753–768, 2006.
4. M. Eichberg, D. Germanus, M. Mezini, L. Mrokon, and T. Schäfer. *Qscope: an Open, Extensible Framework for Measuring Software Projects.* In Proceedings of 10th European Conference on Software Mainte- nance and Reengineering (CSMR), pp. 111–120. IEEE Computer Society, 2006.
5. M. Eichberg, S. Kloppenburg, M. Mezini, and T. Schuh. *Incremental Confined Types Analysis.* In Proceedings of the Sixth Workshop

on Language Descriptions, Tools and Applications (LDTA), Electronic Notes in Theoretical Computer Science, pp. 81–96. Elsevier, 2006.

6. M. Eichberg, M. Haupt, M. Mezini, and T. Schäfer. *Comprehensive Software Understanding with Sextant.* In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM), pp. 315–324. IEEE Computer Society, 2005.

7. M. Eichberg, T. Schäfer, and M. Mezini. *Using Annotations to Check Structural Properties of Classes.* In Proceedings of Fundamental Approaches to Software Engineering: 8th International Conference (FASE), Volume 3442 of Lecture Notes in Computer Science, pp. 237–252. Springer, 2005.

8. M. Eichberg. *BAT2XML: Xml-based Java Bytecode Representation.* In Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode), Volume 141 of Electronic Notes in Theoretical Computer Science, pp. 93–107. Elsevier, 2005.

9. M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. *Xirc: A kernel for cross-artifact information engineering in software development environments.* In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE), pp. 182–191. IEEE Computer Society, 2004.

10. M. Eichberg, M. Mezini, and K. Ostermann. *Pointcuts as functional queries.* In Proceedings of Programming Languages and Systems: Second Asian Symposium (APLAS), Volume 3302 of Lecture Notes in Computer Science, pp. 366–381. Springer, 2004.

11. M. Eichberg, M. Mezini, T. Schäfer, C. Beringer, and K.-M. Hamel. *Enforcing System-wide Properties.* In Proceedings of the 2004 Australian Software Engineering Conference (ASWEC), pp. 158–167. IEEE Computer Society, 2004.

1.3 Structure of this Thesis

This thesis is comprised of four parts: an introductory part (I.), two main parts (II. and III.) and a part (IV.) which summarizes the thesis.

I. The introductory part is comprised of this chapter and Chapter 2. This part introduces open static analysis platforms by discussing the motivation behind them, the problem they address and the requirements imposed on them.

   Chapter 2 particularly discusses those requirements that are relevant for facilitating the incremental build process integration of software comprehension and static code analysis tools.

II. This part consists of three chapters that describe the foundations as well as the design and implementation of the prototypical platform Magellan. As we will see, all major requirements identified in Chapter 2 are met by Magellan.

   The first chapter (Chapter 3) proposes an approach to modularizing static analyses such that explicit coupling between them is avoided. This is a prerequisite for open static analysis platforms.

   The architecture of Magellan is presented in Chapter 4. Magellan is an open static analysis platform tightly integrated with the incremental build process of the Eclipse IDE. Magellan implements the approach proposed in Chapter 3.

   Chapter 5 discusses how to embed query engines into Magellan. Two concrete examples will be presented: the embedding of an XQuery engine and a Prolog system.

III. In the second part, several applications of Magellan are presented. This part evaluates various aspects of Magellan. In particular questions related to the feasibility, versatility and the performance of the platform will be answered.[2] Furthermore, the advantages of building different tools on top of a common platform will be emphasized.

   The implementation of Java and XQuery based analyses is discussed in Chapter 6. In particular analyses of structural properties and checkers using intra-procedural control-flow and data-flow information are considered. Based on the performance measurements taken while running the analyses, the overall performance of the platform will be assessed. Furthermore, an estimation of the number of static analyses that can be executed along with the incremental build process is made. Finally, the performance of analyses implemented in XQuery and Java is compared.

[2]The chapters of this part (chapters 6–9) are self-contained and can be read in any order.

The software exploration tool SEXTANT is presented in Chapter 7. SEXTANT enables the user to navigate along different relations between software elements, such as, classes, fields or methods. SEXTANT uses the XQuery interface and, hence, enables an assessment of its use in interactive environments. Furthermore, SEXTANT demonstrates that MAGELLAN facilitates the development of software exploration (comprehension) tools.

Chapter 8 discusses the metrics tool QSCOPE. As SEXTANT, QSCOPE also uses the XQuery interface. But, unlike SEXTANT, the queries (to calculate the metrics) usually analyze all or nearly all project artifacts. Hence, QSCOPE enables an assessment of the suitability of the XQuery engine for analyzing large data sets.

Chapter 9 discusses the implementation of advanced type systems using (a) Java and (b) Prolog. In particular, the implementation of confined types using both technologies is presented. By comparing both implementations the advantages and disadvantages of both approaches are evaluated.

IV. The last part concludes this thesis by summarizing its major contributions and by giving an outlook to future work.

Chapter 2

Requirements on Open Static Analysis Platforms

Jackson and Rinard [JR00] foresee a bright future for (a) sound and complete analyses as well as (b) unsound and incomplete analyses. *Sound* means that every true error is reported and *complete* that no false positives are reported. An example of a tool of the first category is Java PathFinder [HP00]. Examples of tools of the second category are JLint [KARW04], CoffeStrainer [Bok99], IRC [EMS+04], or the tools described in [Cop06, ECCH01, EGHT94, EL02, FLL+02, HP04, Joh79, RSS+04a, GYF06].

The advantage of using sound and complete analyses, i.e., software verification, is that software can be proven to be error free w.r.t. specific properties, such as, stack overflow errors or synchronization errors. The disadvantage of verification is that a formal specification is needed that requires specially trained experts. Moreover, verification tools can often only be applied to code that adheres to severe restrictions; e.g., Java PathFinder [HP00] requires that the state space must be finite and tractable.

Due to these limitations, verification is (currently) only used for mission-critical software and not for enterprise applications. Furthermore, verification introduces another level of complexity in the software development process and does not primarily aim to improve a developer's productivity, which is a target of this thesis. Hence, verification tools are not further considered in the following.

When using unsound and incomplete tools no correctness guarantees can be given and, hence, these tools can also be ineffective. In general, tools that use static analysis are ineffective if the number of false positives is too high, too many errors are not identified, the quality of the error reports is

low (e.g., when it is hard to decide if a report is a false positive or not), the analyses take too much time, or the effort for using the tool is too high. For example, the authors of ESC/Java [FLL+02], where annotations can be used to support the checking process, admit that the effort of annotating a certain application was not justified; annotating the project took three weeks, but only half a dozen errors were found.

Nevertheless, tools that use static analysis are often effective in practice, i.e., capable of detecting a reasonable number of problems in a limited amount of time without requiring the user to have knowledge in static analysis. Hence, these tools can help to improve the quality of the software when applied on a regularly basis. In the following, the term *lightweight static analyses* is used to refer to unsound and incomplete analyses as well as analyses that target structural properties.

lightweight static analyses

Besides lightweight static analysis based tools, which support developers in their day-to-day work by detecting and preventing errors, software comprehension tools also aim to improve the productivity of developers. Comprehension tools foster the understanding of program code by visualizing the system or by providing means to explore the software.

Both categories of tools, i.e., software comprehension tools and static code analysis tools, require the same core functionality such as source code parsers for a variety of different languages, class hierarchy analyses, call graph analyses or query engines. Hence, the development of a common platform is promising to lead to a cross-fertilization between those tools. Base functionality developed as part of a specific project can be reused in other projects.

For example, software exploration tools often only implement very basic analyses because the focus of the tools (researchers) is on providing innovative user interfaces and visualizations. Nevertheless, these tools could profit from more advanced base analyses to provide end-users with richer sets of exploration and visualization possibilities, e.g., to enable the user to navigate to those places in code where a field is potentially initialized. If an exploration tool is built on top of a common platform, its developer can reuse advanced analyses while staying focused on the exploration layer.

A study covering existing tools was carried out to determine the requirements on platforms that should simultaneously serve as a foundation for both categories of tools. The requirements are the result of:

- a comprehensive study of tools that are used to statically find and prevent errors: CoffeStrainer [Bok99], PMD [Cop06, Har05], AspectJ [Lad03, SY02], ESC/Java [FLL+02], Xgcc [ECCH01, AE02, HCXE02],

RacerX [EA03], Saber [RSS+04a, RSS+04b], SLAM [BR02], SPLint [EL02], CodeQuest [HVdM06], PQL [LL05, MLL05], Checklipse [Liv05], CheckJ2EE [Liv04], Findbugs [HP04], JLint [KARW04, AH04], PREfast [Mir04], PREFix [BPS00], Hammurapi[Vla06].

- an analysis of the static analysis platform Aristotle [HR97].
- the aspects of software visualization tools that were identified in the survey by Bassil and Keller [BK01].
- an analysis of well-known software comprehension tools (SCTs): CodeCrawler [DDL99], SHriMP [SWFM97], Searchable Bookshelf [SCHC99], Spool [RSK00], JQuery[JD03], Rigi [MTW93], G^{SEE} [Fav01], HY+ [MS95], Dali [KC98], Ciao [CFKW95], FEAT [RM02], TkSee [SLVA97], and Class Blueprint [DL05]

  A result of the evaluation of these comprehension tools was that all tools follow the overall architecture described in [Lan03]. That is, the tools have either a two or three layered architecture; in case of a two layered architecture the bottom and middle layers are merged. The bottom layer (*metamodel*) stores the information about the software to be analyzed and provides querying capabilities. The middle layer (*core*) defines the tool's domain model and implements the core functionality. The top layer (*visualization layer*) provides the visualization.

  W.r.t. the identified architecture of software comprehension tools only requirements related to the metamodel were taken into consideration. This layer's functionality is independent of the tool's specific comprehension features and, hence, can be provided by a generic platform; as argued by Lanza [Lan03]: *"The metamodel can be developed by someone else ... the [software comprehension] tool provider should not have to write a parser by himslef."* For example, tools as diverse as SHriMP [SM95] and CodeCrawler [DDL99] could be implemented using the same metamodel. Both tools require the same type of information about source elements and their dependencies, e.g., subtypes, supertypes, callers and callees. The provided visualizations, nevertheless, vary widely: SHriMP uses nested graph based visualizations for documenting software structures. CodeCrawler uses metrics based visualizations to foster program understanding.

The features and limitations of the presented tools were identified to derive the requirements on a platform that (a) can serve as a foundation

for code analysis and code comprehension tools, and (b) provides services commonly required by software engineering tools. That is, a platform that fulfills the identified requirements will have to either implement the necessary functionality on its own or at least provide services that facilitate the implementation of the requirements. Hence, when compared to implementing comprehension and analysis tools from scratch, such a platform promises to reduce the necessary effort.

checkers

In the following, analyses that (a) check that a specific property holds, (b) do not derive information used by subsequent analyses, and (c) do not modify the database are called *checkers*. For example, an analysis that checks that the return value of Java's String.concat(...) method is not ignored is called a checker. Analyses that derive information meant to be used by subsequent analyses (or checkers) are called *base analyses*. Examples of base analyses are control-flow and data-flow analyses. The information derived by these analyses is, by itself, rarely interesting for the user.[1] The term *analysis* is used to refer to checkers as well as base analyses.

base analyses

analysis

The identified requirements are grouped in three categories:

Applicability

These requirements are concerned with the applicability of the platform for different purposes, such as, using it as a foundation for lightweight static analyses or software exploration and visualization tools. If one of these requirements is not met, the platform is limited in the types of tools that can be built on top of it.

Scalability

These requirements are concerned with the scalability of the platform and its support for implementing scalable analyses. In case that requirements of this category are not fulfilled, the size of projects that can be analyzed will be smaller.

Usability

These requirements are related to usability issues. For example, if the platform is integrated into an IDE, many tool adoption issues [FES03], such as, "the developer has to learn to use yet another tool", can be avoided.

[1] A similar distinction is also made by Jia and Skevoulis in [JS99]. They distinguish between *generic analyses*, i.e. base analysis, and *specialized analyses*, i.e. checkers.

In the following, each identified requirement will first be explained and then work related to the particular requirement is discussed.

2.1 Applicability

To serve as a foundation for a wide range of different software engineering tools, a platform has to fulfill the following five requirements.

OSAP-R1 Extensible base analyses stack

In this thesis, an analysis stack (a) determines the set of analyses that are executed when analyzing a software, and (b) also determines the order of execution of those analyses. If it is extensible, it is possible to remove or replace existing analyses or to add further analyses.

For example, the analysis stack could be comprised of an analysis which reads in Java class files and a second analysis which is executed thereafter that calculates the intra-procedural control flow graphs (CFG). If the stack is extensible, it is then possible to add a new analysis that calculates data-flow information using the CFG. The provided data-flow information can subsequently be used by checkers to pinpoint developers to issues found in the code.

The implication of a platform that does not provide an extensible base analysis stack is that the set of possible analyses is restricted by the richness of the platform's built-in base model. For example, if the base model does not store information about the call graph, each analysis has to derive this information on its own, if possible at all.[2] Even if each analysis can derive the necessary data, it is still more efficient to implement and to run a corresponding analysis exactly once.

Related Work

In [SLVA97] Singer et al. examined software engineering work practices related to software exploration tools. As part of this work they identified the requirement that platforms should have extensible analysis stacks to enable the integration of special-purpose analysis tools. The program-analysis platform Aristotle [HR97] features an extensible base analysis stack, which

[2]Some tools only support the definition of new checkers in a tool specific language that does not facilitate the definition of arbitrary analyses, e.g. [LL05].

facilitates the addition of new analyses. In Aristotle new analyses are implemented in C and can use information stored in Aristotle's database. Analyses can also store new information in the database.

In FindBugs [HP04] the object graph generated by the BCEL bytecode toolkit [BCE06] is used as the base representation for detecting bug patterns. This representation is close to a one-to-one representation of Java bytecode and sufficient for the implementation of many bug pattern detectors. However, an analysis that requires higher-level information, such as data-flow information, has to derive the information on its own. If several analyses require the same kind of information, it is either derived again and again by each analysis or additional functionality — unrelated to the analysis problem at hand — need to be developed by developers of analyses to control the interaction between the analyses. Both solutions are inefficient and not scalable. The same reasoning applies to PMD [Cop06, Har05] and CoffeeStrainer [Bok99], both tools support the programmatic specification of new analyses, but provide no explicit mechanism to control the interaction between mutually dependent checkers.

Hammurapi [Vla06] features an extensible base analysis stack. New checks are defined as Java rules that are evaluated using a forward chaining rules engine. The results of the evaluation of a rule (checker) can be used by subsequent rules (checkers).

OSAP-R2 Support for open base representations

The term base representation is used to refer to the source model derived by base analyses. The base representation is used by checkers and code comprehension tools to derive information relevant for end-users.

A platform supports open base representations if analyses are allowed to extend and — in particular — to modify the representation derived by previously executed analyses. For illustration, consider the code shown in Figure 2.1. A control flow analysis of the code results in the control flow

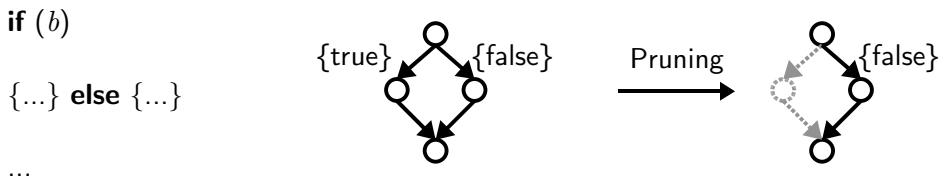

Figure 2.1: Pruning of impossible control-flow paths

graph (CFG) shown in the middle of the figure. This CFG might be accurate enough for many analyses, but a more precise CFG may be useful, e.g., to reduce the number of false positives. Using a data-flow analysis it might be possible to find out that b is always false and, thus, to prune the left (true) path of the CFG as wells as the corresponding code. This results in the control flow graph on the right-hand side.

After that, however, it is useless to execute an analysis that reports dead code to the developer — no error report will ever be generated. It might even be misleading if the analysis is executed and no errors are reported. The developer would probably get the wrong impression that there is no dead code. A platform that supports open base representations has to be able to identify and handle such cases; it has to detect analyses that are incompatible or that need to be scheduled in a very specific order. In the given example, it has to execute the dead code checker before the analysis that removes the dead code. [3]

Related Work

Open base representations, which enable analyses to transform (modify) the results of previous analyses, are generally not supported by existing tools. However, the need for open base representations can be derived from the observations made in [AH04, ECCH01, EL02].

In these papers the authors write that the requirements on the base analyses may rise during the development of checkers and — more important — that it is often impossible to determine upfront the exact kind of analyses needed to effectively check code until the checkers are actually run and their results are evaluated.

For example, in [ECCH01] the authors write that they could reduce the number of false warnings by ∼18% by just pruning simple, impossible paths. But, before actually using their checkers they were not aware of this fact. In general, support of open base representations is required to handle the

[3] This requirement is meaningless for platforms that do not fulfill the extensible base analysis stack requirement (OSAP-R1) since analyses that contribute to the base representation are not supported at all. However, platforms that have an extensible base analysis stack must not necessarily support open base representations. Such platforms can specify that additional base analyses are only allowed to derive additional information and that the analyses are not allowed to manipulate the information derived by previous analyses. Such platforms are, however, not scalable as keeping all information is prohibitive in terms of memory usage. Hammurapi [Vla06] is an example of a static analysis platform that has an extensible base analysis stack, but which does not have an open base representation.

conflicting requirements, between analyses that depend on a specific code representation and other analyses that depend on a transformed variant of the representation.

OSAP-R3 Enabling cross-artifact reasoning

Enabling cross-artifact reasoning means that analyses can take different kinds of resources into consideration, e.g., Java source code, properties files and XML documents.

The semantic of todays software is often not determined by program code alone. Information defined in other artifacts of a software development project, such as XML deployment descriptors, also determine the runtime semantics of the application. Hence, when analyzing a project's resources it is not sufficient to consider the code only. For example, in Enterprise Java Beans [EJB03] projects, the transactional behavior of the application can be defined programmatically in Java code, or declaratively in the descriptor of the bean. Both artifact types have to be taken into consideration when checking that the transactional behavior of the application is well-defined.

Related Work

Facilitating cross-artifact reasoning is generally recognized as important. Kazman and Carriere [KC98] explicity mention that tools for understanding software architectures require an open approach to information extraction. For example, the software exploration tool G^{SEE} [Fav02, Fav01] explicitly supports multi-source exploration. G^{SEE}'s exploration environment is independent of the source of data and virtually any kind of structured data is supported. For each type of resource, a so-called backpacker needs to be implemented, a small component with a standardized interface that facilitates the exploration of a specific type of resources.

The program-analysis platform Aristotle [HR97] supports multiple programming languages to facilitate cross-language analyses. Saber [RSS+04a, RSS+04b] is an example of a static code analysis tool that also analyzes a fixed set of different types of resources, namely Java source code and Java Server Pages (JSPs).

OSAP-R4 Support for parameterized checkers

Parameterized checkers can be instantiated by end-users to define application / project / company specific checks.

Though, it is possible to develop and build into a platform a variety of project independent checkers that can be used out of the box to check large parts of many projects, these checks are not sufficient. In case of modern software projects additional support is needed, e.g., for checking and enforcing restrictions concerning the project specific use of methods, fields and types.

Such checkers, however, cannot be provided in a ready-to-use way. Furthermore, it is also not reasonable to expect enterprise applications developers to implement checkers directly on top of the base representation using a low-level API. Instead, it is necessary to enable the definition of project-specific analyses as instantiations of predefined templates to make the specification of new checkers as easy as possible. As written in [Bok99], requiring a user to implement checkers at the programming language level is adequate only for the most complex checkers.

A typical example of a project dependent analysis is one that reasons about method call protocols. Given the following class it is desirable to check that terminate and close are only called after open was called and that a closed or terminated connection is not reopened.

```
1 class Connection {
2   /** Opens a connection, reopening connections is not possible. */
3   public void open() {...}
4
5   /** Sends all remaining data and then closes the connection. */
6   public void close() {...}
7
8   /** Immediately closes the connection. */
9   public void terminate() {...}
10 }
```

Listing 2.1: Implicit declaration of a method call protocol

Related Work

For recurring patterns Saber [RSS+04a, RSS+04b] implements a fixed set of rules which need to be parameterized before being used. A rule in Saber is, e.g., “Must call X after Y”; after specifying concrete values for X and Y, the code can be analyzed to detect corresponding violations. At this level,

only minimal knowledge is required to derive customized analyses. RacerX [EA03], FCL [HHR04], and the tool described in [ECCH01] all demonstrate that a large number of useful analyses can be defined at a high-level by instantiating predefined analysis.

OSAP-R5 Enabling the embedding of query engines

Query engines facilitate the declarative specification of analyses, e.g., using Prolog. They are an indispensable prerequisite when developing software comprehension tools or end-user extensible tools.

Since parameterized checkers (OSAP-R4) can only be defined for the most common cases, it is necessary to provide means for the declarative specification of analyses to enable users less familiar with static analysis to define new analyses. As discussed in [HCXE02], the definition of many application specific analyses can be simplified when compared with directly using the API of the platform.

When using query languages the user has to specify "what to check" and not "how to check". However, the choice of the query language is extremely important as pointed out in [JD03]: *"...the logic language was hard to use for complex queries. This is true even for developers reasonably familiar with the query language."* Hence, it is important to enable the embedding of a variety of different engines.

The following requirements on query engines were identified:

Semantic queries enable to search for semantic elements and not only for the occurrence of a specific string in a set of artifacts. A prototypical example is: "Find all **classes** which **inherit** Serializable."

Search tools such as grep, which just search for character sequences, are not sufficient [SCHC99].

Query chaining means that the result(s) of a query can be used as the starting point for subsequent queries. For example, after executing a query that returns all methods that access a specific field, it should be possible to execute a second query that returns the set of the declaring classes of those previously identified methods. Query chaining is particularly required by software exploration tools to implement a step-by-step exploration process.

Query filtering refers to the ability to filter those queries that are not applicable in a specific context. Support for binding queries to specific types of elements for which the query can be executed is required. E.g., the context of the query: "Get all declared fields" are classes. Hence, the platform should provide a mechanism to determine whether a query can be evaluated in a given context.

Automatic incrementalization of queries means that after a change to a subset of the project's artifacts the result of a query is updated and that the query is not reevaluated for the whole program. Given the source model before the change and a description of the latest changes, only those parts of the source model are (re)analyzed that are related to the changes. Supporting automatic incrementalization is indispensable for queries that are executed as part of the incremental build process.

Related Work

In general, declarative (query) languages are widely used by software engineering tools for the definition of new analyses. For example, PMD [Cop06] enables the user to define new analyses using XPath, in PQL [LL05] the user defines a pattern that is transformed into Datalog, and in [LRY+04] parametric regular path queries are proposed. Logic query languages are used by ASTLOG [Cre97], JQuery [JD03] and CodeQuest [HVdM06]. In Xgcc the code is analyzed and

In the context of software comprehension tools, the need for semantic based browsing facilities (semantic queries) was identified by Singer et al. in [SLVA97]. They require that support for browsing the full spectrum of semantic items is necessary. A semantic item is each character sequence that has special semantics w.r.t. the type of artifact and the local context in which the sequence occurs. Examples of semantic items are the names of classes and methods, or an EJB's remote interface name. This requirement was also identified by Sim et al. [SCHC99], they write that *"a search facility for a software architecture must be able to specify searches for meaningful elements in the source code such as functions and variables"*.

Query chaining is pioneered by software exploration tools, such as, JQuery [JD03] and Ciao [CFKW95]. In Ciao [CFKW95], the result of executing a query is a virtual database on top of which so-called operators and views operate. An operator provides means to create more complex abstractions and the result is again a virtual database. Ciao filters queries by restricting

the set of operators that the user can execute to those that are legal given the database on top of which the operator should operate.

The automatic incrementalization of Prolog based queries and its application to static analysis is discussed in [SR06].

2.2 Scalability

Regarding scalability, the following three requirements were identified.[4]

OSAP-R6 Execution of required analyses only

Given a set of analyses, the platform should be able to automatically determine those analyses that need to be carried out and those analyses that are not required given the current configuration. This serves to reduce the overhead caused by the analyses. E.g., if none of the user configured analyses requires control-dependence information, a corresponding analysis should not be carried out, even if installed.

Related Work

Most static analysis based tools enable the user to configure the set of checkers that should be executed [HP04, Int06a]. But, the internal base analyses are always run independently of the user-selected checkers. However, given that most existing tools are domain-specific, e.g., detecting race conditions and deadlocks [EA03, AH04], or security vulnerabilites [LL05], running all analyses is not a concern; the results of all base analyses are required as soon as a single checker is used. That is, this requirement is unique to open static analysis platforms.

OSAP-R7 Support for incremental analyses

[4] A general requirement, such as, "Support for the analysis of large programs" is meaningless as it is impossible to determine when such a requirement is fulfilled. In [RSS+04a] a program of 8770 classes — $\sim$ 2 million lines of code (LOC), with roughly 250 LOC per class in average — is considered to be large. In [EA03] a program of 500K LOC is considered to be large, and in [KARW04] applications are considered to be large that have around 100.000 LOC. Finally, in [Liv04] a suite of multiple applications with together 130.000 LOC is said to be a suite of large applications.

If analyses should be executed as part of the incremental build process, it is important that the implementation of incremental analyses is supported. That is, the platform should provide the base representation of the program and the set of changes such that the analyses can determine the impact of the changes on the representation and maintain the information derived by them.

Related Work

This requirement is derived from the observation that most tools that analyze a project as a whole, require too much time to let them run regularly as part of the incremental build process. For instance, Saber [RSS+04a] and PQL [LL05] both require several seconds or even minutes to analyze a project.

The most recent release of the IDEA IDE [Int06a] features a set of non-trivial incremental analyses that detect bugs and bug patterns related to violations of inter-class relations.

CodeQuest [HVdM06] is a code querying tool for Java programs. It is tightly integrated with the Eclipse IDE's incremental build process and incrementally updates its underlying SQL database. However, the evaluation of analyses (SQL queries) does not happen as part of the build process. Rather, they are evaluated on explicit user demand between two builds.

OSAP-R8 Support for meta-analyses

Meta-analyses enable the efficient execution of several analyses at once. They do not modify the base representation or generate any other output themselves. Instead of having each individual analysis traverse the base representation, a meta-analysis traverses the structure once and calls back the other analyses whenever necessary, basically following the visitor design pattern [GHJV95] with analyses as visitors. This way, meta-analyses enable the efficient execution of larger numbers of analyses.

For example, the Enterprise JavaBeans specification [EJB03] defines a large number of implementation restrictions, which have to be followed by enterprise components. Many of these restrictions basically require that the implementing classes have specific structural properties, e.g., "Enterprise bean classes must be defined as public...". The checkers of these properties have to analyze the same elements, i.e., class, method and field declarations. By grouping these checkers by means of a meta-analysis redundant traversals of the same data structure can be avoided. Concerning such checkers, meta-

analyses are an enabling technique that makes it possible to execute a larger number of them as part of an incremental build process.

Enabling the time efficient execution of checkers that detect violations that affect the application start-up is of particular importance. The sole incentive to use such checkers is to prevent the failing of a lengthy application start-up during testing. The same applies to analyses that detect errors that prevent the successful deployment of components.

Related Work

Meta-analyses are used in: CoffeeStrainer [Bok99], FindBugs [HP04], Xgcc [HCXE02], PMD [Cop06] and PREfast [Mir04]. All these tools use a visitor like approach to improve the analysis time of groups of similar checkers; i.e., the tools traverse their underlying source models and call back the checkers when needed.

2.3 Usability

Regarding the usability of open static analysis platforms, the following requirements were identified.

OSAP-R9 Integrated into an IDE

Todays software projects are usually developed using sophisticated integrated software development environments (IDEs) such as IDEA [Int06a] or Eclipse [Ecl06]. IDEs support developers in coping with the complexity of software projects by providing functionality to compile, debug and browse a project's artifacts. Hence, platforms for static analyses should be integrated with IDEs to leverage the existing infrastructure and to provide the most benefit for developers. IDE integration enables developers to put the results of analyses (error messages and visualizations) directly into relation with the project's artifacts and, thus, facilitates reasoning about the project.

As written in [AH04], being able to put the results of checkers into relation with the project artifacts is essential for understanding the cause of a bug report. In case of checkers an integration enables developers to navigate from bug reports to the source document(s) and possibly to further documents to comprehend the bug report.

Concerning IDE integration two aspects were identified:

Support for compile-time and on-demand analyses

Compile-time analyses run continuously on-the-fly as part of the incremental build process offered by modern IDEs; on-demand analyses are performed as an explicit step initiated by the developer.

For compile-time analyses, there is no need for the developer to explicitly start the analysis process — the latter is automatically triggered when a developer makes changes to a project's artifacts. Though continuous checking is generally desirable, it is only applicable for reasonably fast analyses. For example, many checkers that enforce structural properties fall into this category [HP04].

To enable users to execute analyses that are too slow to run them regularly as part of the incremental build process, the execution of analyses on demand has to be supported. On-demand analyses also have to be supported to make the development of software comprehension tools possible. A comprehension tool's analyses are executed when requested by the user.

Supporting on-demand analyses requires that tools developed on top of the platform can specify the type of information to be available between two builds. For example, if a comprehension tool specifies that it requires a representation of Java classes using the bytecode toolkit BAT [BAT06], then the platform (a) has to schedule the necessary analyses that make the corresponding representation available and (b) has to make sure that no other subsequently executed analysis modifies or transforms the representation such that it no longer satisfies the requirements of the tool.

Configuration sharing

This requirement concerns team support. It is important that the configuration of checkers including project-dependent ones can be shared among developers. This enables, e.g., a project lead to setup and share the configuration of the analyses that have to be passed before code can be checked in a version control system.

Related Work

The need for IDE integrated browsing facilities was identified by Singer et al. in [SLVA97]. They require that IDEs should support browsing a project's sources. Basically they demand an IDE integration of software comprehension tools.

Most modern static code analysis tools are now integrated into IDEs and enable the evaluation of checkers on-demand, e.g., FindBugs [HP04], Checklipse [Liv05], PMD [Cop06] and Saber [RSS+04a].

However, an integration with the incremental build process is usually lacking. AspectJ [Asp06], which facilitates the definition of simple checkers by means of declare warning and declare error statements, is an exception. AspectJ supports incremental compilation and, hence, the corresponding declare warnings and error messages are also incrementally evaluated. However, AspectJ is not a static analysis tool and the set of errors and warnings that can be defined is rather limited.

Though FindBugs [HP04] is also integrated into the incremental build process, it does not correctly update error reports where the underlying analysis depends on inter-class information, e.g., type hierarchy related information. Findbugs lacks a mechanism that determines the effect of a source code change on inter-class relations. The current release of the IDEA IDE [Int06a] features analyses that can be evaluated as part of the IDE's incremental build process.

OSAP-R10 Configurable set of base analyses

The set of required base analyses depends not only on the needs of the checkers, but also on the requirements of a user on the precision of the analyses. In some cases a trade-off between a more precise analysis and a faster analysis is possible. In these cases the user should be able to choose the base analyses most appropriate for the checking task at hand.

For example, the user should be able to choose between a context-sensitive and a context-insensitive points-to analysis. If a user is willing or able to spend more time for the analysis to get better results this should be possible.

Related Work

This requirement is derived from the observation that it is impossible to

(automatically) predict the effect of certain base analyses on the quality of a checker. As pointed out in [Mir04]: *" ... variations in coding styles also cause variations in what is reported."* Hence, given a specific coding style, some base analyses might be ineffective. In this context, effective refers to the ratio between the additional overhead caused by the more precise base analysis and the number of false warnings not generated by checkers later on. In other words, an analysis that runs a long time, but which suppresses only a very small number of false warnings is ineffective and the user should be able to choose a less precise analysis.

OSAP-R11 Error report management

Some analyses tend to produce large numbers of false warnings. For instance, in [AH04] 198 false positives for a program with 7500 lines of code were identified and in [AB01] "several thousand" false warnings were identified. In case of Safe [GYF06], one checker even had a false positive rate of nearly 100% for a specific project.

This problem can be addressed by running more sophisticated analyses that do not report so many false warnings. For instance, PREFix [BPS00] has only a false warning rate between 10% and 25% of all generated warnings. However, highly sophisticated analyses are often much slower and can require several minutes or even hours. This is a serious limiting factor for the integration of analyses' into an incremental build process. Hence, it can be more efficient to combine a fast and more error prone analysis with an effective error management than to run an analysis only irregularly.

With respect to handling error reports the following requirements were identified:

Error reports with predefined severity levels

For checkers that do not generate false positives, it is sufficient to associate a simple severity level with each message, e.g.: error, warning or info. A category of checkers for which predefined severity levels are often well suited are those that check for violations of structural properties. These checkers do not produce false positives [HP04].

This simple mechanism is also sufficient for checkers which produce false positives, but where each false positive is an indicative of at least a serious violation of a best practice. Consider a checker to find violations of the rule: "The finalize method should always call super.finalize()".

Let's assume that the checker finds a finalize method that does not call super.finalize() and where no superclass (except of java.lang.Object) implements the method. A report of this finding would be a false warning, because the finalize method of java.lang.Object does not perform any special action. So, not calling the super method does not lead to any problem. However, it is serious enough to always generate a warning message: If a superclass later on implements finalize, e.g., to dispose some system resources, this method will not be called, eventually resulting in a resource leak.

Dynamic ranking of error reports

The platform should support dynamic ranking of errors based on the properties of each report. Dynamic ranking means that the order in which messages are presented to the user is not predefined; it is rather determined based on the properties of each report. The goal of the ranking is to direct the user to those reports which most likely describe real errors. This requirement basically concerns checkers with a high likelihood of reporting false warnings.

For example, checkers to find deadlocks and data races are prone to generate false warnings. For these checkers it is possible to use the length of the call chain that would lead to the error as the basis for calculating the rank. The reasoning is that the underlying analyses have only limited precision and, hence, the likeliness of a false positive increases with the number of involved analysis steps. A report of a possible deadlock that results from the analysis of a long call chain with multiple threads involved is more likely to be a false positive than a report based on a very short path.

Management of the history of error reports

One possibility to suppress false warnings is to manually identify a warning as a false positive once and to use this knowledge to suppress the generation of the message in the future. For example, it is possible to store the kind of error and its relative source location in a method to get robust information that can be used to suppress the error message in the future.

Filtering of error reports

Filters provide an effective means to suppress large numbers of false warnings. For example, domain knowledge or the location of an error can be used to filter false positives. An example of the latter case are

errors related to code defined in an API or framework not relevant for the current application.

Graphical error reports

To facilitate comprehension of complex errors, e.g., as identified by tools that detect race conditions and deadlocks, developers often need additional control- and data-flow information. For example, comprehending a deadlock warning without giving a detailed call graph is hard for many non-trivial examples. Hence, it is necessary that checkers can generate reports which include control-flow and data-flow information and that the platform provides appropriate means to visualize this information.

Related Work

Static ranking of error reports is widely used [Vla06, HP04, Lad03, SY02].

Dynamic ranking of errors is used by RacerX [EA03] and xgcc [HCXE02]. Engler and Ashcraft [EA03] propose to use the length of the call chain that would lead to the error as the basis for calculating the rank.

A history of warning reports is used by xgcc [HCXE02] to suppress false warnings. Xgcc stores for each false warning the file name, the name of the function, and the name of the variables involved; this information is used to suppress the corresponding warning in the future.

Sophisticated filtering mechanisms, beyond simple filters to suppress all messages of a specific checker or set of checkers, are provided by Saber [RSS+04b]. E.g., in [RSS+04a] warnings related to a class DriverManager are filtered, because the erroneous code is related to the graphical user interface (GUI) and will never be executed as part of the analyzed server side application.

As identified in [AH04, EA03, HP00], reports related to complex errors, such as, race conditions or deadlocks, require that the data- and control-flow information that led to the report are presented to the user. Without these information comprehending the report is hard for many non-trivial examples. The path relevant for a warning is reported by SLAM [BR02]. Saber [RSS+04a] and PREFix[BPS00] also represent data-flow information.

Applicability	
OSAP-R1	**Extensible base analyses stack**
OSAP-R2	**Support for open base representations**
OSAP-R3	**Enabling cross-artifact reasoning**
OSAP-R4	**Support for parameterized checkers**
OSAP-R5	**Enabling the embedding of query engines**
Scalability	
OSAP-R6	**Execution of required analyses only**
OSAP-R7	**Support for incremental analyses**
OSAP-R8	**Support for meta-analyses**
Usability	
OSAP-R9	**Integrated into an IDE**
OSAP-R10	**Configurable set of base analyses**
OSAP-R11	**Error report management**

Table 2.1: Requirements on open static analysis platforms

2.4 Conclusions

In this chapter, requirements on open static analysis platforms were discussed. The requirements are the result of an analysis of existing code analysis and code comprehension tools with regard to the development of a common platform that supports both types of tools.

During the study, it became evident that the requirements of static code analysis tools on a common platform are more homogenous than those of code comprehension tools. Nevertheless, the requirements of comprehension tools on the back-end are comparable and, thus, by fulfilling those requirements it is still possible to significantly support comprehension tools. In case of code comprehension tools the user interfaces, however, differ widely, whereas the user interfaces of code analysis tools are basically identical. Hence, providing comprehensive support for static code analysis tools is easier than supporting code comprehension tools.

Overall, the eleven requirements summarized in Table 2.1 were identified. The requirements: “support for parameterized checkers“, “support for meta-

analyses" and "error report management" are primarily the result of the analysis of static code analysis tools that detect issues in the source code. The requirements to support query filtering and query chaining — both part of the "enabling the embedding of query engines" requirement — were identified while analyzing code comprehension / code exploration tools.

In the following part of this thesis, concepts and techniques will be presented that enable the development of platforms that fulfill the identified requirements.

Part II

Magellan: an Open Static Analysis Platform

Chapter 3

An Approach to Decoupling Analyses

Any problem in computer science can be solved with another layer of indirection.

David Wheeler

Part of the material in this chapter is published in: *Integrating and Scheduling an Open Set of Static Analyses [EMK+06].*

3.1 Introduction

To a varying degree static analyses are used in the back-ends of tools for finding errors [RSS+04a, FLL+02, Joh79], type checkers [EKMS06, FL03], and for visualizing [DDL99, MS95] as well as exploring [SWFM97, SCHC99, JD03] software systems. Traditionally, these tools are developed independently as standalone tools. But, to further improve the productivity of the development process more and more of these tools are now integrated into IDEs [HP04, RSS+04a, Mir04] and a few tools are even integrated with the incremental build process [HP04, Int06a, HVdM06].

If, however, multiple independent tools are integrated with the incremental build process valuable time and memory is wasted. Each tool maintains its own source model, even though, the requirements of the tools on the

source model overlap widely and large parts of the source model could be shared. Besides being an engineering issue — the same functionality for parsing and analyzing code is developed over and over again — this waste of processing time and main memory limits the number of tools that can be used simultaneously and the size of the projects that can be analyzed.

To address these issues an approach is proposed that facilitates an efficient integration and scheduling of an open set of static analyses. The individual analyses are decoupled and the execution of the analyses is coordinated such that the overall time and space consumption is minimized. As part of the minimization of the overall analysis time, the approach also identifies possibilities for the automatic parallelization of analyses. Parallelization is necessary to make efficient use of modern multi-core / multi-processor architectures. Further, the approach enables the user to select those analyses that are needed in the context of the developer's project. Only the selected analyses as well as any analysis that provides information required by a selected one is run as part of the incremental build process.

To facilitate the decoupling of analyses, the effect of an analysis is specified w.r.t. an open data model. Each analysis specifies the data it reads and contributes to the source model. This, in turn, requires means of coordination between analyses that write and read the data model. E.g., a call graph analysis would specify that it reads a specific source code representation (provided by another analysis) and that it derives the call graph.

Before continuing the discussion of the platform, the statements made in the pervious paragraphs are reconsidered in terms of the sample analyses shown in Table 3.1, along with the data they depend on. Though this discussion focusses on analyses for finding programming errors and bad smells, it equally applies to analyses used by software comprehension tools.

Table 3.1 illustrates that static analyses differ widely in the data they require, but they also share subsets of data. For example, both the SA and the CFT analysis require data flow information. Each analysis could of course compute all the data it requires from the raw source code or from a generic representation of the project. However, implementing and running several instances of an algorithm for data flow analysis wastes both engineering effort and computational resources. Furthermore, it is a waste of resources to reify a generic representation of the entire software when the analyses consume only information about a part of the project. For example, the EH analysis requires only information about the interfaces of Java classes; method bodies or other artifacts such as deployment descriptors are irrelevant.

To cope with the issues stated in the previous paragraph, analyses are

ID	Description	Required Data
NSF	Searches for finalize methods that do not call super. finalize.	control flow graph (CFG)
EH	Searches for Java classes overriding either equals(Object) or hashCode(), but not both.	interfaces of Java classes
SA	Searches calls of String.append where the return value is ignored.	data flow information (method implementation)
CTAV	Searches for Enterprise Java Beans that use declarative and programmatic transaction demarcation [EJB03].	type hierarchy, method implementation, EJB deployment descriptors
CFT	Realization of Confined Types [EKMS06] based on Java annotations.	type hierarchy, type hierarchy changes, data flow information, public interfaces of libraries

Table 3.1: Sample analyses and the data they depend on

divided into small modular producer-consumer units. Analyses such as SA and CFT can share the results produced by a base analysis for data flow information; similarly, EH can consume the results of an analysis that produces information about the interfaces of Java classes only. This requires that analyses are run in a well-defined order to satisfy their dependency relations.

These relations cannot, however, be expressed by a total order, since the set of analyses is open. It is also desirable to automatically select and run only analyses that produce information consumed by those analyses directly selected by the user. A base analysis, e.g., for getting the type hierarchy, should only run if its result is needed by a user selected analysis.

The dependencies cannot be represented by a partial order graph either. For better performance, analyses should be able to transform and modify existing analysis data instead of generating new data. Furthermore, several analyses that generate the same information can co-exist within the platform and it should be ensured that at most one of them is run. Both cases are not expressible by a partial order.

To coordinate the execution of a set of analyses their dependencies are mapped to a constraint system. By solving the constraint system, an order in which the analyses can be executed is determined. The coordination

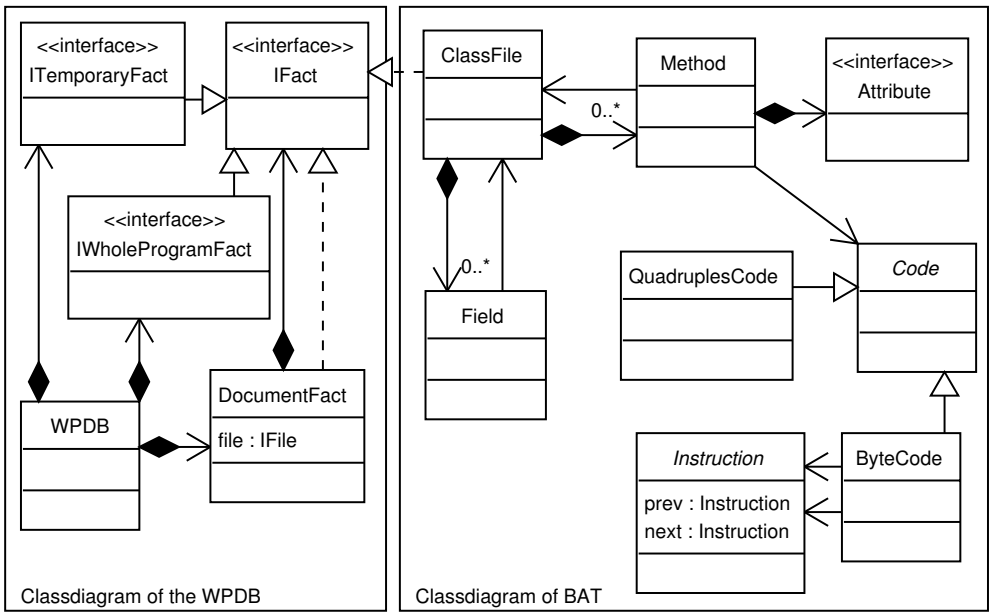

Figure 3.1: Combined class diagram of the WPDB and BAT

unit, which is called the scheduler, treats analyses as modules that write, read or maintain the source model. Each analysis describes its properties and dependencies in a special analysis specification language (ASL). These specifications are mapped onto a constraint system which is fed to a constraint solver. Adding an objective function to the set of constraints allows to calculate a schedule that is optimal with regard to the number of analyses to run.

The rest of this chapter is structured as follows: In Section 3.2 the data model is described. After that, it is shown how to specify an analysis' dependencies in Section 3.3 and how to calculate a schedule in Section 3.4. Section 3.5 makes an assessment of the approach w.r.t. the requirements identified in Chapter 2. Section 3.6 concludes this chapter by summarizing the model and its properties.

3.2 The Analysis Data Model

whole-program database (WPDB)

The analysis data in our platform is stored in the *whole-program database* (WPDB). The WPDB is an object graph built-up cooperatively by the executed analyses. The WPDB has a set of designated root objects which are

called *facts*. The architecture of the fact objects is shown within the box on the left hand side of Figure 3.1, entitled "Class diagram of the WPDB". The WPDB aggregates three types of facts:

Document Facts: For each resource (file) in the project an object of class DocumentFact (see Figure 3.1) is created. The document fact always keeps a reference to the underlying file. A document fact enables analyses to attach derived information by means of classes that implement the IFact interface. For example, a representation of a Java class file is a fact that is typically attached to a document fact. If the Java Bytecode Analysis Toolkit BAT [BAT06] is used to represent Java class files, instances of the class ClassFile — within the box in the right-hand side of Figure 3.1 — are created to store information about the individual Java class files.

A ClassFile object stores the name of the class, the class's modifiers, information about declared annotations and the implemented interfaces. Furthermore, a ClassFile object keeps references to the set of declared methods and the set of declared fields. Each field is represented by a Field object and each method is represented by a Method object (see Figure 3.1). A method's implementation is either represented using a byte code based representation ByteCode or using a higher-level quadruples based representation QuadruplesCode.

A document fact is automatically created, added to, and removed from the database corresponding to the type of action on the underlying file. Further, the set of all document facts that are added, created or removed from the database in a build is also directly made available to the analyses. This enables analyses which can work incrementally per document to process only the delta to the previous build.

Whole Program Facts: Information that cannot directly be associated with specific documents is stored in the database using *whole program facts*. A whole program fact always needs to be maintained by the analysis that creates it. After a full build, the analysis has to re-create the whole program fact; after an incremental build, the analysis has to bring the information up-to-date to reflect the current project's state. For example, an analysis that makes the type hierarchy information available has to update the type hierarchy whenever the developer makes a change that affects the type hierarchy.

Temporary Facts: Information that is only valid during a build step is stored in *temporary facts*. All temporary facts are automatically deleted before each build. For example, a type hierarchy analysis could also generate information about the changes to the type hierarchy for the benefit of subsequent analyses. However, this information is only valid for the current build.

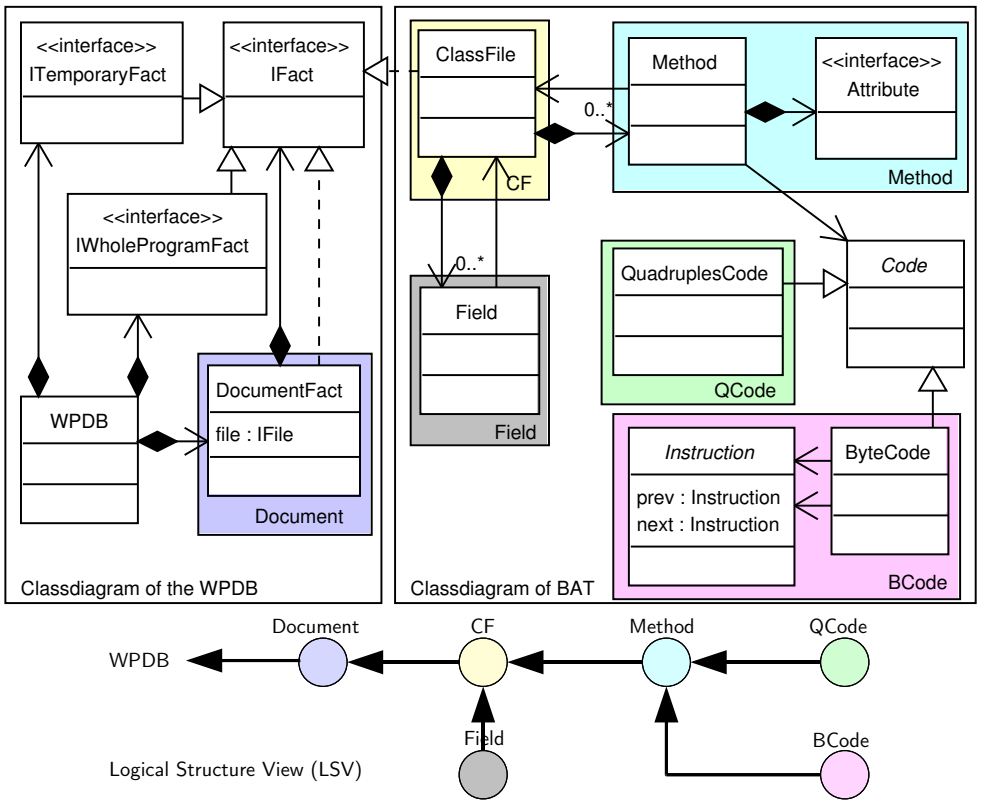

Figure 3.2: The LSV of the WPDB

logical structure view (LSV)

Data dependencies in the WPDB are expressed in the *logical structure view* (LSV) — a directed acyclic graph. Every node in the LSV stands for a group of WPDB elements. A group can be a selection of objects or a selection of (primitive) field values of the objects in the WPDB. We call nodes in the LSV *entities*. The lower part of Figure 3.2 shows an extract of the LSV. Also, its mapping to the corresponding parts of the WPDB is shown by the boxes around elements of the WPDB and BAT class diagrams.

entities

Consider for an example the box labeled "Method" surrounding the class Method and Attribute in the upper right corner of the BAT class diagram.

This boxing states that a LSV method entity is mapped to a WPDB method and all its attributes.

We refer to entities in the LSV by using paths starting at the WPDB vertex (Figure 3.2). The WPDB vertex is, however, never used in path expressions, it is always implicitly assumed. For instance, to refer to the BCode entity we write: **Document**/CF/Method/BCode.

Edges in the LSV express data dependencies as defined in Figure 3.3. For

---

Let v be an arbitrary LSV entity and w an entity that is dependent on v, i.e., a path from w to v exists in the LSV.

- If a WPDB element is changed that is represented by v then every other element that is affected by this change is represented either by v or by w. Hence, WPDB elements that are affected by a change have to be represented by the same LSV entity as the changed element (v) or by an LSV entity (w) that is dependent on the LSV entity of the changed element (v).
- Declaring an entity w as dependent on an entity v implies that there are no conflicts between an analysis that changes the data associated to w or any of its dependent entities and those that just read the data associated to v.
- Analyses that access siblings do not conflict.

Figure 3.3: Semantics of the dependencies between LSV entities

example, `Field` and `Method` are declared as dependent entities of `CF`. Hence, an invalidation of the information on a class entity automatically invalidates information on its fields and methods. But, there are no conflicts between analyses that process `Field` and `Method` entities respectively.

The properties of the LSV are leveraged by the scheduler to parallelize analysis executions.

The part of the LSV that is relevant for scheduling the analyses is reconstructed from the set of analysis specifications, as detailed in Section 3.4.

However, there is a trade-off to be considered when designing the LSV: A fine-grained LSV increases the possibilities for parallelization, but decreases the ease of describing and understanding the dependencies among elements in the database.

The mapping between the WPDB and the LSV is specified as part of the libraries that are used to represent the data in the WPDB. E.g., the

mapping of WPDB elements to the LSV is specified in an extra document delivered with the part of the BAT library that manages the information about Java class files (see Figure 3.1).[1] All analyses that make use of a specific library (e.g., BAT) then have to adhere to the library's specification of the LSV to the WPDB mapping, i.e., the analyses must use the specified LSV entities and the same path statements to refer to the entities. When creating the mapping all dependencies between LSV entities have to have the semantic as defined in Figure 3.3. The mapping specification also defines which information is represented by an entity w.r.t. the project's artifacts. For example, as part of the definition of the CF fact it is stated that for each class file of the project an instance of a ClassFile object exists. This instance is associated with the DocumentFact that represents the ".class" file (cf. Figure 3.2).

To extend the LSV and WPDB, for example to make the intra-procedural control-dependence graphs (CDG) of methods available, the user first needs to determine where to store the information in the physical model. BAT's representation of class files, for example, enables to attach arbitrary information to Method and Code objects by means of Attributes (see Figure 3.2). An attribute is a simple container object to store further information. Hence, after calculating the CDG of a method, the CDG could be stored as an attribute of the analyzed Code element. Further, a new LSV entity CDG is declared that is dependent on, e.g., the BCode entity. This new entity can then be used by analyses to declare their dependencies.

3.3 Specifications of Analysis Dependencies

The *analysis specification language* (ASL) is used to declare the data required and provided by each analysis in terms of the logical structure view described in the previous section.

The ASL supports six types of dependencies as shown in the ASL grammar in Figure 3.4. Listing 3.1-3.3 illustrate the specification of the sample analyses from Table 3.1.

```
1 analysis CFP (* creates class file representation *)
2   writes Document/CF, Document/CF/Field, Document/CF/Method,
        Document/CF/Method/BCode
```

[1] Though, this mapping is specified informally, an approach that would use Java annotations or a similar technique and would then enable a semi-automatic derivation of the LSV is easily imaginable.

ASL	::= **analysis** ID STATEMENT*
STATEMENT	::= DEPENDENCY PATH*
DEPENDENCY	::= **reads-global** | **reads** | **writes** | **invalidates** | **maintains** | **writes-temporary**
PATH	::= ID [/ PATH]

Figure 3.4: The ASL grammar

```
3
4 analysis DDP (* creates EJB deployment descriptor representation *)
5   writes Document/EJBDD
```

Listing 3.1: Analyses that make base information available

```
1 analysis BCFG (* creates the control-flow graph (CFG) *)
2   writes Document/CF/Method/BCode/CFG
3
4 analysis BtoQ  (* transforms the bytecode in 3-address SSA form *)
5   invalidates Document/CF/Method/BCode
6   writes Document/CF/Method/QCode
7
8 analysis LIB  (* maintains the repository of used library classes *)
9   reads Document/CF/Method/BCode
10  reads-global Document/CF
11  maintains Library/CF/Field_NON_PRIVATE, Library/CF/
        Method_NON_PRIVATE
12
13 analysis TH  (* maintains the type hierarchy *)
14  reads-global Document/CF, Library/CF
15  writes-temporary TypeHierarchyChange
16  maintains TypeHierarchy
17
18 analysis CTA1  (* prog. and decl. transaction demarcation is used *)
19  reads Document/EJBDD
20  reads-global TypeHierarchy, Document/CF/Method/BCode
21  writes CTAViolations
22
23 analysis CTA2  (* alternative CTA analysis *)
24  reads Document/EJBDD
25  reads-global TypeHierarchy, Document/CF/Method/QCode
26  writes CTAViolations
```

Listing 3.2: Base analyses that read, create and transform the source model

```
1 analysis NSF (* finalize does not call super.finalize() *)
2   reads Document/CF/Method/QCode/CFG
3
4 analysis EH (* equals and hashcode have to be implemented pairwise *)
5   reads Document/CF/Method
6
7 analysis SA (* String.concat() must not be ignored *)
8   reads Document/CF/Method/QCode
9
10 analysis CFT (* realizes Confined Types *)
11   reads TypeHierarchyChange
12   reads-global TypeHierarchy, Document/CF/Method/QCode,Library/CF/
       Method_NON_PRIVATE
13
14 analysis CTAV (* wraps CTA and CTA2 *)
15   reads CTAViolations
```

Listing 3.3: Analyses that just read the database (Checkers)

A **reads** dependency on some LSV entities means that the analysis works incrementally on the specified input data. For example, the EH checker (Listing 3.3, Line 4) specifies that it reads the entities referred to by the path expression **Document**/CF/Method. A **reads–global** dependency, on the other hand, means that the analysis needs data of the specified kind for *all* documents, not just those processed in the current build. The current implementation of the type hierarchy analysis, e.g., needs access to all class files, not just those changed; hence, the corresponding **reads–global** dependency in Listing 3.2, Line 14.

A **writes** dependency specifies that the analysis provides data of the specified type for documents that are changed in the current build step only. For example, the DDP analysis (Listing 3.1, Line 5) specifies that it writes the EJBDD entity and implicitly reads the preceding entities, i.e. the **Document** entity. In general, a write dependency **writes** $e_1/e_2/../e_{n-1}/e_n$ specifies that the entities $e_i, \forall i = 1..n-1$ are read and that only the entity e_n is written. If an analysis specifies a writes dependency with multiple paths, e.g., as the CFP analysis shown in Listing 3.1, then only those elements are treated

as read that are not declared to be written by any path. Hence, only the Document entity is read by CFP.

A **writes–temporary** dependency is used for data that is automatically invalidated (and hence removed by the platform) before the next build. For example, the type hierarchy analysis (Listing 3.2, Line 13) also provides information about changes to the type hierarchy between the current and the previous build. Since this information is only valid for one specific build step, it is declared using **writes–temporary**. As in case of **writes**, only the last entity of the path is written and the previous entities are read.

The **invalidates** dependency specifies that after executing the analysis the last entity referred to by the given path expression as well as all entities depending on it are no longer valid. This is usually the case if an analysis provides its result by transforming existing data in the WPDB. For example, the analysis which transforms a method's bytecode representation into the 3-address based representation (Listing 3.2, Line 4) changes the existing data in the WPDB. Hence, it specifies that the BCode entity will become invalid when the analysis is executed.

Finally, **maintains** is used by an analysis to declare that it creates an entity and updates it during the following builds. For example, the type hierarchy analysis declares to maintain (Listing 3.2, Line 16) the TypeHierarchy entity. Again, only the last entity is considered to be maintained.

Analyses may overlap in both their input and output data. If multiple analyses produce the same data, the scheduler decides which of these analyses will be executed. There can also be multiple analysis specifications for the same analysis to express that an analysis can use different data as input. For example, the checker for detecting conflicting transaction demarcations (CTAV - Listing 3.3, Line 14) needs either the byte code (BCode - Listing 3.2, Line 18) or the SSA-transformed code (QCode - Listing 3.2, Line 23), hence there are two specifications for this analysis. Such alternatives give the scheduler more leeway in scheduling an analysis.

An analysis specification also serves as a contract on what the analysis implementation is allowed to do with the WPDB. The result of an analysis may only depend on data in the WPDB whose entity in the LSV is read. The analysis must not add any data to WPDB entities which are not marked as **writes** or **writes–temporary** nor change any data that is not marked as **invalidates** or **maintains**, respectively.

3.4 Scheduling Analyses

3.4.1 Processing the Analyses Specifications

To calculate an execution schedule for a set of analyses, their ASL specifications are mapped onto a constraint system, which is solved by means of integer programming.

The first step towards this mapping is to reconstruct the logical structure view from ASL specifications. For this purpose, each ASL statement is parsed and a new entity is created for each path element that is not yet represented in the LSV. The special entity for **Document** is included by default. Moreover, each entity is directly connected with its parent entity. For example, for the path statement **Document**/CF/Method two additional entities are generated: one for CF and one for Method; the entity for Method is made a dependent entity of the CF entity.

Once the LSV is generated, it is recorded for each entity which analyses access it and how. This information is needed for the generation of the constraint system. The following six sets are recorded, whereby $\mathcal{A}$ denotes the set of all (installed) analyses a, and $\mathcal{E}$ denotes the set of all entities e in the LSV:

- R denotes the set of analyses that read the entity. It includes any analysis that explicitly states to do so. An analysis is also added to the set R for each entity on paths of its **writes** or **invalidates** statements except of the last entities.

- W denotes the set of analyses that specify **writes** or **writes−temporary** statements for the entity. In case of **writes−temporary**, the entity is marked as **temporary** and it is checked that all dependent entities are also marked as **temporary**. At runtime temporary entities will automatically be deleted before an incremental build.

- I denotes the set of analyses that directly invalidate an entity. An analysis a invalidates an entity e in two cases: (1) a explicitly declares e in an **invalidates** statement, (2) a **reads** e and directly invalidates some other entity, on which e depends.

- I^{P} denotes the set of analyses that implicitly invalidate the entity e. An analysis a implicitly invalidates an entity e, if a neither **reads** nor directly **invalidates** e, but declares to invalidate an entity, on which e directly or indirectly depends.

- R^G denotes the set of analyses that specify a **reads−global** statement for the entity; i.e., the analysis requires access to the currently added documents as well as documents that have been processed in an earlier build.

- M denotes the set of analyses that maintain the information of the entity.

In Figure 3.5, an example of an LSV is depicted that shows which analyses access an entity and how they access it. The LSV is the result of analyzing the ASLs of the previously discussed analyses: CFP (Listing 3.1), BCFG (Listing 3.2) and BtoQ (Listing 3.2).

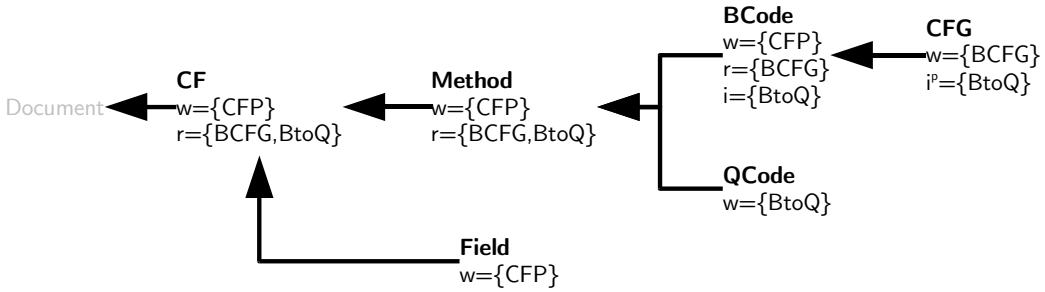

Figure 3.5: Example of an LSV-access-tree

The CFP analysis, which processes Java class files and provides an initial representation of the bytecode, **writes** the entities CF, Field, Method and BCode. The BCFG analysis, which processes a method's byte code to calculate the CFG specifies a **reads** dependency on CF, Method and BCode entities and a **writes** dependency on the CFG entity. The Field or QCode entities are, however, not accessed. The BtoQ analysis which transforms a method's bytecode representation into a quadruples code representation explicitly **invalidates** the BCode entity and **writes** the QCode entity. Since the analysis does not read the dependent CFG entity, the CFG entity is implicitly invalidated; i.e. if the CFG is available before the execution of the BtoQ analysis it is no longer available afterwards.

3.4.2 Generating the Constraint System

Based on the LSV-access-tree, the constraint system is generated to calculate the schedule. The constraint system ensures that every calculated schedule

is valid — in fact, the constraint system can be seen as a declarative specification of the semantics of the ASL. A schedule is valid if all requirements of all analyses are met:

- The entities an analysis specifies to read were made available in a previous step and are not (yet) invalidated.
- A dependent entity is available only if the parent is also available.
- Every entity is made available at most once.
- Every entity is explicitly invalidated at most once.

The constraints ensure that an analysis that writes an entity is guaranteed to have exclusive access to the entity and race conditions cannot occur. If the constraints have no solution, an error is reported.

In the following, the process of generating the constraint system is presented. In doing so, the following variables are used:

- T_a^S, $a \in \mathcal{A}$ denotes the point in time (execution step) in a schedule S, at which an analysis a is executed. $T_a^S = 0$ means that a is not scheduled.
- V_e^S, $e \in \mathcal{E}$ denotes the point in time at which e becomes valid. $V_e^S = 0$ means that e will never be available.
- I_e^S, $e \in \mathcal{E}$ denotes the point in time at which e becomes invalid. $V_e^S > 0 \wedge I_e^S = 0$ means that e is available during the next build.

The generated constraints make use of the following definitions: For any entity e the functions $w(e), m(e), r(e), r^g(e), i(e), i^p(e)$ return the sets W, M, R, $\mathrm{R^G}$, I, and $\mathrm{I^P}$ of the entity e respectively. Given an entity e, the predicate $isTemporary(e)$ returns *true* if e is marked as temporary and *false* otherwise.

The range of the variables must be bound in order to solve the constraint system using integer programming, e.g., using ZIMPL [Koc04] / lp_solve [BEN05]. The domain of the variables T_a^S, V_e^S and I_e^S is $[0, ..., MAX]$ where MAX is $2 * m + n$ ($m = |\mathcal{E}|$ being the number of entities and $n = |\mathcal{A}|$ being the number of installed analyses). MAX defines the theoretical maximum value of the variables T_a^S, V_e^S, and I_e^S. To schedule n analyses that process m entities, we need at most $2 * m + n$ time slots. $2 * m$, because each entity e is associated with two time slots: V_e^S and I_e^S. This covers the worst-case where all analyses are executed sequentially, all analyses create only one entity, the analyses do not conflict and entities are also invalidated.

$$V^S_{\mathsf{Doc}} = 1 \wedge I^S_{\mathsf{Doc}} = 0 \tag{3.1}$$

Availability (validation) of entities:

for each $e \in (\mathcal{E} - \{\text{Doc}\})$:

$$V^S_e > 0 \Rightarrow \sum_{a \in (w(e) \cup m(e))} T^S_a > 0, \tag{3.2}$$

for each $e \in \mathcal{E}$:

$$\forall a \in (w(e) \cup m(e)), T^S_a > 0 \Rightarrow \sum_{x \in (w(e) \cup m(e))} T^S_x = T^S_a \tag{3.3}$$

$$\forall a \in (w(e) \cup m(e)), T^S_a > 0 \Rightarrow T^S_a + 1 = V^S_e \tag{3.4}$$

$$\forall a \in (r(e) \cup r^g(e)), T^S_a > 0 \Rightarrow 0 < V^S_e < T^S_a \tag{3.5}$$

Invalidation of entities:

for each $e \in \mathcal{E}$:

$$\forall a \in (r^g(e) \cup m(e)), T^S_a > 0 \Rightarrow I^S_e = 0 \tag{3.6}$$

$$isTemporary(e) \Rightarrow V^S_e \leq I^S_e \tag{3.7}$$

$$I^S_e > 0 \Rightarrow 0 < V^S_e < I^S_e \tag{3.8}$$

$$\forall a \in i(e), T^S_a > 0 \Rightarrow I^S_e = T^S_a \wedge \sum_{x \in i(e)} T^S_x = T^S_a \tag{3.9}$$

$$\forall a \in i^p(e), T^S_a > 0 \wedge V^S_e > 0 \Rightarrow 0 < I^S_e < T^S_a \tag{3.10}$$

$$\forall a \in (r(e) - i(e)), T^S_a > 0 \wedge I^S_e > 0 \Rightarrow T^S_a < I^S_e \tag{3.11}$$

Objective function:

$$minimize \left(\sum_{a \in \mathcal{A}} T^S_a + \sum_{e \in \mathcal{E}} V^S_e \right) \tag{3.12}$$

Figure 3.6: Constraint system for calculating an analysis schedule

The constraints are shown in Figure 3.6 and their purpose is explained in the following.

V^S_{Doc} and I^S_{Doc} (Equation 3.1) are the variables for the special **Document** entity. The **Document** entity is — by definition — available at the very beginning of the schedule ($V^S_{\mathsf{Doc}} = 1$) and must not be invalidated ($I^S_{\mathsf{Doc}} = 0$).

Implication (3.2) requires that — except for the document entity which is provided by the framework — every entity that becomes available during the analysis process is actually created by an analysis. The constraint ensures that at least one analysis is scheduled that writes e. The implication (3.3) ensures that an entity is created at most once. Implication (3.4) defines that a specific entity e is available in the step immediately following an analysis that writes e and (3.5) specifies that an entity e is available before an analysis is executed that reads e. Hence, (3.4) and (3.5) ensure the correct order between analyses that write and read an entity.

Implication (3.6) enforces that entities that will be (re-)read or maintained during the following build are not invalidated. For an entity that is marked as temporary, constraint (3.7) ensures, that a point in time can be determined at which the entity can become invalid.

Constraint (3.8) ensures that only entities are invalidated that were created previously. Constraint (3.9) enforces that only one analysis explicitly invalidates an entity. Furthermore, the entity is invalidated in the same step as the analysis that invalidates the entity, to make sure that no other analyses are executed in parallel that read the entity.

Constraint (3.10) states the relation between the execution time of an analysis a and the invalidation time of entities that are implicitly invalidated by a. If an entity e that is implicitly invalidated by a is valid ($V^S_e > 0$), then it is just required that e is no longer valid after the execution of the analysis. It is, however, not required that an implicitly invalidated entity is explicitly invalidate by a. This allows another analysis executed before a to explicitly invalidate e.

Constraint (3.11) specifies that an analysis need to be executed before any entities become invalid that are read by the analysis.

The objective function (3.12) is the minimum of the sum of all analysis times and the availability times of entities. Minimizing the sum of the analyses times is equivalent to finding a schedule that executes only necessary analyses as early as possible. By including the points in time at which entities become available it is ensured that those analyses are scheduled that create the minimum number of entities necessary for satisfying all constraints.

If we directly solve the constraint system in Figure 3.6, no analysis is scheduled; the T_a^S values for all analyses will be zero as this minimizes the objective function. To calculate a schedule, for any user selected analysis a we add the constraint:

$$T_a^S > 0 \tag{3.13}$$

In addition to analyses that are automatically executed as part of each incremental build, support for tools is required that run analyses on-demand of the user, i.e., between two incremental builds. To support on-demand analyses the MAGELLAN scheduler provides an interface that can be used to specify the entities that need to be available between two incremental builds. For example, a software comprehension tool that operates on BAT's code representations would call the scheduler and specify a dependency to the **Document**/CF/Field and Document/CF/Method entities. To enforce the inter-build availability of a specific entity e, the scheduler just adds the following constraint to the constraint system:

$$V_e^S > 0 \wedge I_e^S = 0 \tag{3.14}$$

This constraint ensures that the corresponding entity is available between two builds and that the tool's analyses will not fail due to missing data.

3.4.3 Example

Step										
1	2	3	4	5	6	7	8	9	10	11
V_{Doc}	T_{CFP}	V_{CF} V_{F} V_{M} V_{BC}	T_{BCFG} T_{LIB} T_{EH}	V_{CFG} V_{L} V_{LCF} V_{LF} V_{LM}	T_{NSF} T_{TH}	V_{THC} V_{THF} I_{CFG}	T_{BtoQ} I_{BC}	V_{QC}	T_{NSF} T_{CFT}	I_{THC}

Table 3.2: Example analysis schedule

Table 3.2 shows an example schedule that is calculated when the user selects all analyses in Listing 3.1-3.3, except for the CTAV analysis (Listing 3.3). For each step, the schedule shows the analysis which has to be executed and the entities which become valid, respectively invalid:

1. the **Document** (Doc) entity becomes valid.
2. the CFP analysis is executed.
3. with the beginning of step 3 the CF, Field (F), Method (M) and, BCode (BC) entities are available.
4. the BCFG analysis can run in parallel with the LIB and the EH analysis.
5. with the beginning of step 5 the CFGs of methods are available. Furthermore, the information about the used libraries (library L, library class file LCF, public fields in the library LF and public methods of the library LM) is also available.
6. the TH and NSF analyses can run in parallel.
7. with the beginning of step 7, the type hierarchy (THF) and the information about type hierarchy changes (THC) are available. Further, the CFG entity is invalidated and, hence, no longer available.
8. the analysis that transforms the method bodies in the 3-address based representation (BtoQ) is executed which directly invalidates the BCode entity (BC).
9. the 3-address based code representation (QC) is available.
10. the CFT analysis and the NSF checker is executed.
11. the type hierarchy change information (THC) can become invalid (it was marked as temporary).

The values of all other variables, i.e., the variables not shown in the schedule, such as e.g., $T_{\mathsf{CTAV}}, I_{\mathsf{CF}}, I_{\mathsf{F}}$, etc. are zero.

3.4.4 Performance

The performance of the scheduling process is briefly evaluated in terms of the number of analyses that can be scheduled in reasonable time. The constraint systems is realized using ZIMPL [Koc04] as the mathematical programming language and lp_solve [BEN05] for solving it. The set of analyses used for the evaluation includes:

- analyses defined in Listing 3.1 – 3.3

- 20 other analyses that check the use of the standard Java API (cf. Appendix V)

- an incremental inter-procedural call-graph analysis, similar to the one described in [SP01]

Including helper analyses defined by the checkers, 66 different analyses are used for the evaluation.

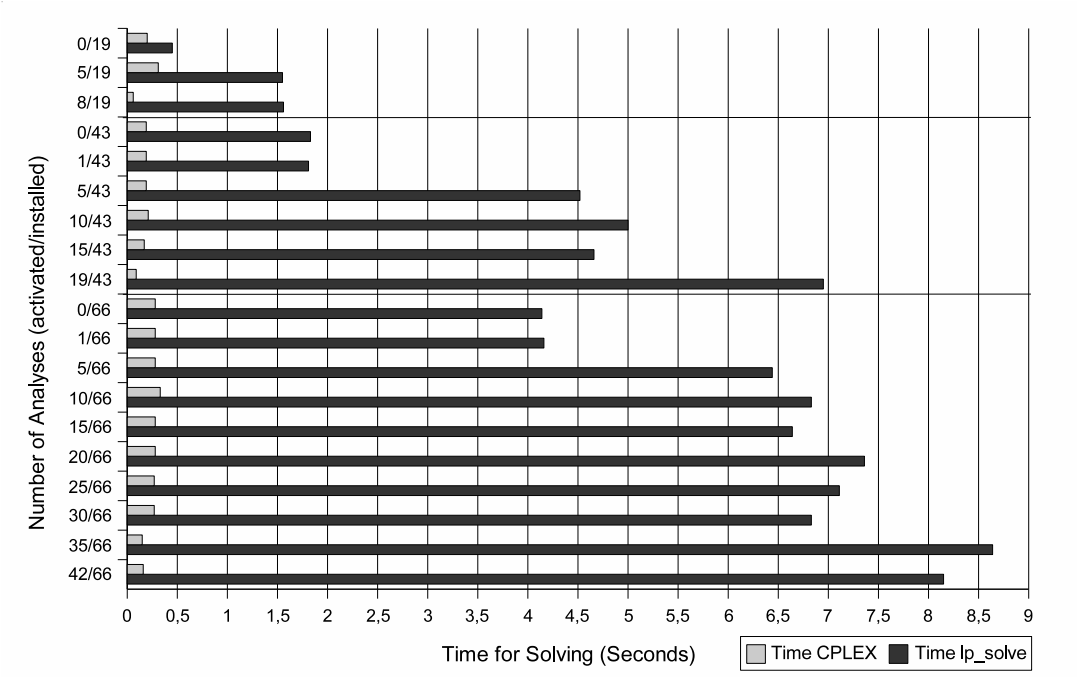

Figure 3.7: Times for calculating analysis schedules

If all checkers are activated, more than 40 analyses will be scheduled; i.e., the value of the T_a^S variables of more than 40 analyses will be larger than 0. This schedule is calculated in less than 10 seconds on a P4/3GHz as shown in Figure 3.7. Using a commercial grade integer programming solver [CPL06] the schedule is even calculated in less than 0.5 seconds and there is only a slight increase in the time to calculate the schedule when more analyses are installed. Hence, the calculation of the schedule is not a limiting factor.[2]

[2]The time for analyzing a project rises along with the project's size, the time for calculating the schedule is, however, independent of the project's size and just depends on the number of installed and selected analyses.

3.5 Evaluation of the Approach

In the following, the proposed approach is evaluated w.r.t. the requirements on open static analysis platforms identified in Chapter 2:

Extensible base analyses stack

At the core of the proposed approach is the modularization and decoupling of analyses to enable an extensible base analyses stack. As the discussed examples throughout this chapters show, this goal is achieved.

Further, the approach also handles incompatible analyses as well as analyses that (partly) derive the same information. Hence, the approach also facilitates the integration of independently developed analyses that might have conflicting requirements.

Support for open base representation

The proposed approach features a lightweight extensible data-model. Only a small core of the data model is predefined. The predefined part includes:

- the types of facts that can be stored in the database: whole program facts, resource based facts and temporary facts.
- how the project's resources are reflected in the database: each resource is represented by exactly one resource based fact.

Other than this predefined part, the data model is open and analyses can read, write and invalidate the data stored in the database.

The approach also enables to specify the data that needs to be available between two incremental builds. Hence, tools that analyze the data stored in the WPDB between two builds are directly supported.

Enabling cross-artifact reasoning

Two features of the data model enable cross-artifact reasoning: (a) the base representation is open, (b) the data model is language neutral.

For example, to analyze relations between different types of artifacts one can write two different analyses, one for each type. The analyses store the respective representations in the WPDB. Any other analysis can then declare corresponding **reads** dependencies and reason about the relations defined between the different types of artifacts (cf. the CTAV analysis from Listing 3.3).

Support for parameterized checkers

At the core level no special provisions need to be taken to support parameterized checkers.[3] To schedule a checker with different instantiations it is sufficient to schedule a dummy checker which has the same requirements as the parameterized checkers. After calculating the schedule the dummy is then replaced by the concrete parameterized instantiations of the checker.

Hence, given the proposed approach supporting parameterized checkers is basically an issue of providing a user interface for instantiating checkers.

Execution of required analyses only

In the presented approach, a schedule is calculated by solving an optimization problem. The objective function minimizes the sum of the number of scheduled analyses and created entities. Hence, base analyses that only derive entities that are neither directly nor indirectly used will never be executed at runtime.

Support for incremental analyses

Incremental analyses are supported at the model level and the implementation level:

Model Level: The effect of each analysis on the underlying database is specified using the primitives of the analysis specification language: **reads**, **writes**, **maintains**, **invalidates** and **reads−global**. Since these primitives were explicitly designed to enable the specification of incremental analyses, they are supported at the model level.

Implementation Level: At execution time an analysis can always access the information which documents were added, removed or changed in the current build. For documents that are removed or changed in the current build the data associated with the old documents is still accessible in the same build step.

Support for meta-analyses

At the model level no special provisions need to be taken to support meta-analyses; only implementation level support is required. Regarding the scheduling process, checkers executed by meta-analyses are treated in the same way as checkers that do not use meta-analyses.

[3]Recall that checkers never modify data stored in the database

The meta-analyses themselves are not scheduled. After calculating the schedule, checkers that use meta-analyses are removed and replaced by their meta-analyses. If checkers that use the same meta-analysis are scheduled at different points in time, a new instance of the meta-analysis is added to the schedule for each of the checker's points in time. After that, the removed checkers are registered with the instance of the meta-analysis that is scheduled at the same point in time. At execution time the meta-analyses will then execute the registered checkers.

For example, assuming that we have three checkers A, B and C that declare that they need to be executed by a meta-analysis M.[4] Further, let's assume that the result of calculating the schedule is that A and B are scheduled in step x and C in step y ($x \neq y$). Given this schedule the checkers A, B and C are removed from the schedule and two instances of the meta-analysis M are scheduled instead: M_x in step x and M_y in step y. After that, the analyses A and B are registered with the instance M_x and C with M_y. At execution time the meta-analyses then execute the analyses A,B and C. Such a schedule, where the same meta-analysis is scheduled several times, can result when the requirements of the analyses, e.g., of A,B and C, differ. If the checkers that declare to use a specific meta-analysis are identical, then they will be scheduled at the same point in time and will be executed by the same meta-analysis.

Configurable set of base analyses

In the proposed approach, users can select base analyses in the same way as checkers (3.13). For illustration, assume that different analyses are available that derive the same kind of information, e.g., a program's call graph [GC01]. Unless the user selects a specific algorithm the current scheduler will chose arbitrarily between the analyses. But, if the user has selected a specific analysis, this one will be selected by the scheduler.

However, imagine a checker that is less likely to generate false warnings if dead code is eliminated, but which can also process code containing dead code. Given that the user has selected the dead code analysis, it would be meaningful to execute the checker after the removal of the dead code. But, this is not supported by the model. A checker (analyses) is always scheduled as early as possible and since the checker

[4] How a checker specifies to be executed by a meta-analysis is unrelated to the concept. However, it is imaginable that such information is specified along with a checker's meta-information.

can process code containing dead code, it will be scheduled before the dead code analysis.

Hence, configurable (base) analyses are only partly supported by the current model.

3.6 Summary

In this chapter, an approach that enables the integration of an open set of static analyses into the incremental build process was proposed. The approach considers analyses as data producers, transformers and consumers. This view enables a decoupling of analyses and facilitates an integration of independently developed analyses.

To determine the order in which to execute a set of analyses, the effect of an analysis on the whole program database is specified, i.e., it is specified which data is added, removed and changed by the analysis in case of incremental builds.[5] An analysis' effect is specified w.r.t. a high-level view on top of the underlying database. The high-level view is a directed acyclic graph that models dependencies in the underlying database such that two parts of the database are independent if no path between the nodes that represent the different parts exist. Two analyses that process independent parts of the database will never conflict.

The specified dependencies are used to derive a constraint system that — when solved — determines the order in which the analyses can be executed. As part of solving the constraint system opportunities for parallelizing analyses are detected. This reduces the overall processing time required to execute all analyses later on.

As discussed in this chapter, platforms implementing the proposed approach can fulfill the requirements regarding the execution of analyses identified in Chapter 2. Also calculating a schedule given a set of analyses will not be a limiting factor as the performance evaluation of the scheduler has shown. Furthermore, the approach enables to detect analyses that can run in parallel and, hence, a more efficient use of computational resources is potentially possible when running on multi-core CPUs or multi-processor systems. Hence, from a theoretical point of view such platforms meet the prerequisites to enable the simultaneous integration of different static analyses along with the incremental build process.

[5] A full build is an incremental build in which all documents are considered changed.

However, it remains to be shown that such a platform is actually feasible and does enable the integration of several analyses along with the incremental build process, in particular the following open issues can be identified:

- With respect to the scalability:
  - What is the overhead caused by the platform during incremental builds?
  - What are the memory requirements for keeping the whole program database in memory?
  - How many analyses can be run as part of the incremental build process?
  - What is the performance gain due to the automatic parallelization of analyses?
- Does such a platform reduce the engineering efforts for developing new static code analyses and software comprehension tools or does the effort for specifying the dependencies and understanding the model overcompensate the gained advantage of not having to implement everything from scratch?

These questions are answered in the following chapters.

Chapter 4

Architecture of Magellan

In this chapter, an overview of the architecture of the open static analysis platform Magellan is given. Magellan implements the approach proposed in Chapter 3 and is tightly integrated with the Eclipse IDE's [Ecl06] incremental build process.

In Section 4.1, the building blocks of Magellan's architecture are presented. The program flow is described in Section 4.2. In Section 4.3, the architecture is evaluated w.r.t. to the requirements identified in Chapter 2. Section 4.4 concludes this chapter by summarizing what is achieved and what needs to be done.

4.1 Building Blocks

The overall architecture of Magellan is depicted in Figure 4.1. The five main building blocks of Magellan (AnalysisRegistry, Scheduler, Dispatcher, UI, WPDB) and their dependencies are explained next.

AnalysisRegistry

The AnalysisRegistry is the central unit where all analyses are registered and managed.

An analysis is registered by using Magellan's registry extension point. An extension point is Eclipse's mechanism to enable a plug-in to specify where other plug-ins may contribute functionality to the plug-in [DFK+04]. Hence, analyses are also implemented as Eclipse plug-ins. But, except from using the extension point mechanism no further dependencies between analyses implemented for Magellan and Eclipse are necessary.

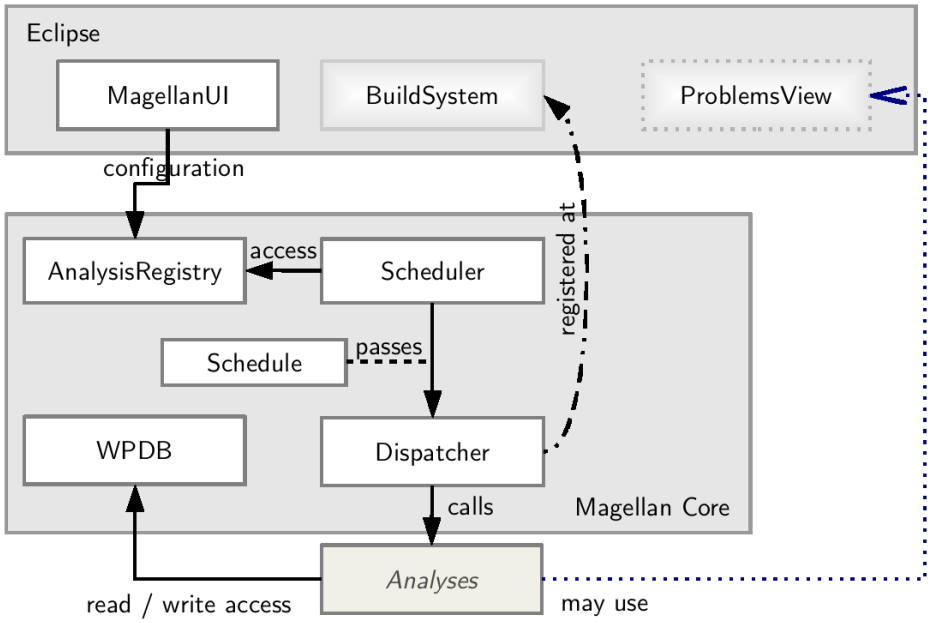

Figure 4.1: Overall architecture of Magellan

To register, e.g., an analysis x.y.Z the analysis' plugin descriptor is as follows:

```
<plugin>
  <extension point="de.tud.magellan.analysis">
    <analysis class="x.y.Z" />
  </extension>
</plugin>
```

The descriptor specifies that it extends the extension point de.tud.magellan.analysis and that the "analysis" is implemented by the class x.y.Z. This class, however, must not implement the analysis on its own; it can also be a wrapper around a query, e.g., written in XQuery or Prolog, that actually implements the analysis. In this case, the class x.y.Z would just provide the Magellan integration.

During startup of the IDE the analysis' plugin descriptor is parsed by Eclipse and then the analysis is registered with Magellan's AnalysisRegistry.

The location of an analysis' specification is specified using Magellan's @ASL annotation as shown in the following listing:

```
@ASL("Z.asl")
public class Z extends IAnalysis {...}
```

When an instance of an analysis is registered with the registry, Java reflection is used to extract the ASL annotation specifying the location of the ASL file.

Scheduler

The scheduler is responsible for calculating the schedule. To do so, it uses the AnalysisRegistry to get information about the configured analyses and their specifications. The calculated schedule is then passed to the dispatcher.

The constraint system is generated using the mathematical programming language ZIMPL [Koc04] and solved using lp_solve [BEN05], as described in Chapter 3.

Dispatcher

The dispatcher executes the schedules. It registers itself with the Eclipse build system. After that, Eclipse will always call the dispatcher when the project or parts of the project have changed and the project needs to be build.

Whole Program Database (WPDB)

The WPDB stores the source model as derived by the executed analyses. The WPDB's implementation is not further detailed as it is a one-to-one implementation of the model proposed in Chapter 3 (cf. Figure 3.1).

MagellanUI

The user interface of Magellan is shown in Figure 4.2. It enables the user to configure the base analyses and the checkers that should be executed as part of the incremental build process. As shown in Figure 4.2, analyses can be grouped in different categories (e.g., "Base Analyses" or "Java API Based Checkers"). The grouping mechanism facilitates comprehension of the purpose of the analyses. Moreover, for each analysis a short description is presented. After activating Magellan for an Eclipse project the project's configuration page is extended to enable the configuration of Magellan. When the user has changed the configuration and presses the Apply or Ok button the scheduler is called to calculate a new schedule. After that, the project is analyzed.

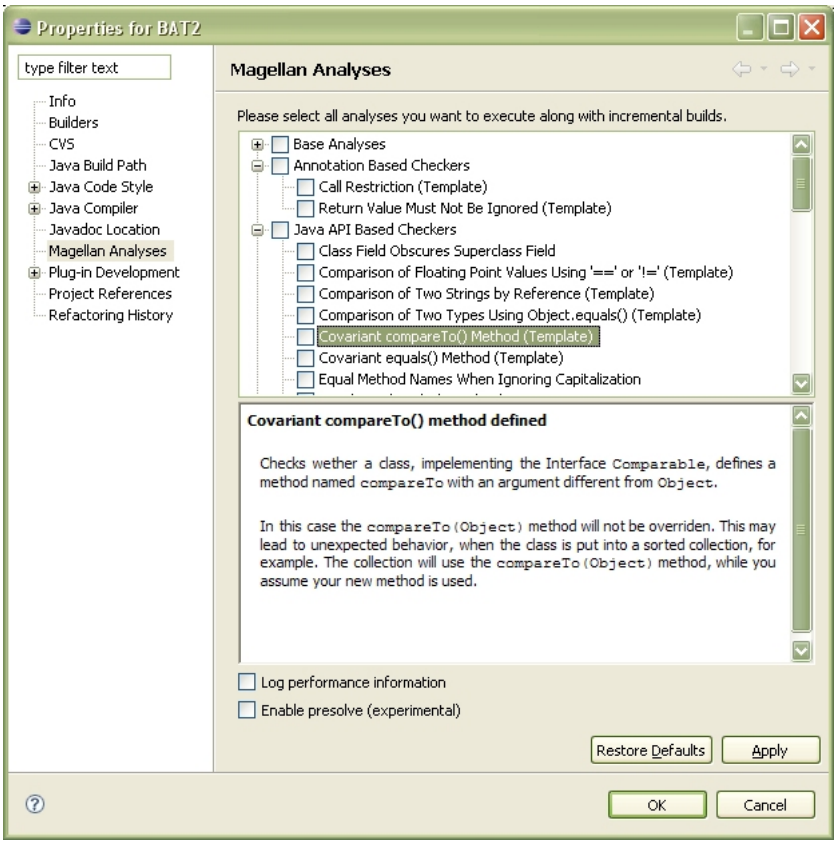

Figure 4.2: The MAGELLAN properties dialog.

Besides providing functionality related to configuring, scheduling and executing an open set of static analyses, no further functionality is implemented as part of MAGELLAN. For example, the checkers that were developed to evaluate MAGELLAN reuse Eclipse's problems view to show the detected violations and errors.

4.2 Program Flow

In the following, the program flow of full builds and incremental builds is explained. A full build is executed if either all project resources have changed, the user explicitly requests it, or if the whole project need to be reanalyzed. An incremental build is executed if a subset of the project's resources has changed. Immediately after activating MAGELLAN for a project, a full build is executed. Hence, a full build always precedes incremental builds. For this reason, full builds are discussed first.

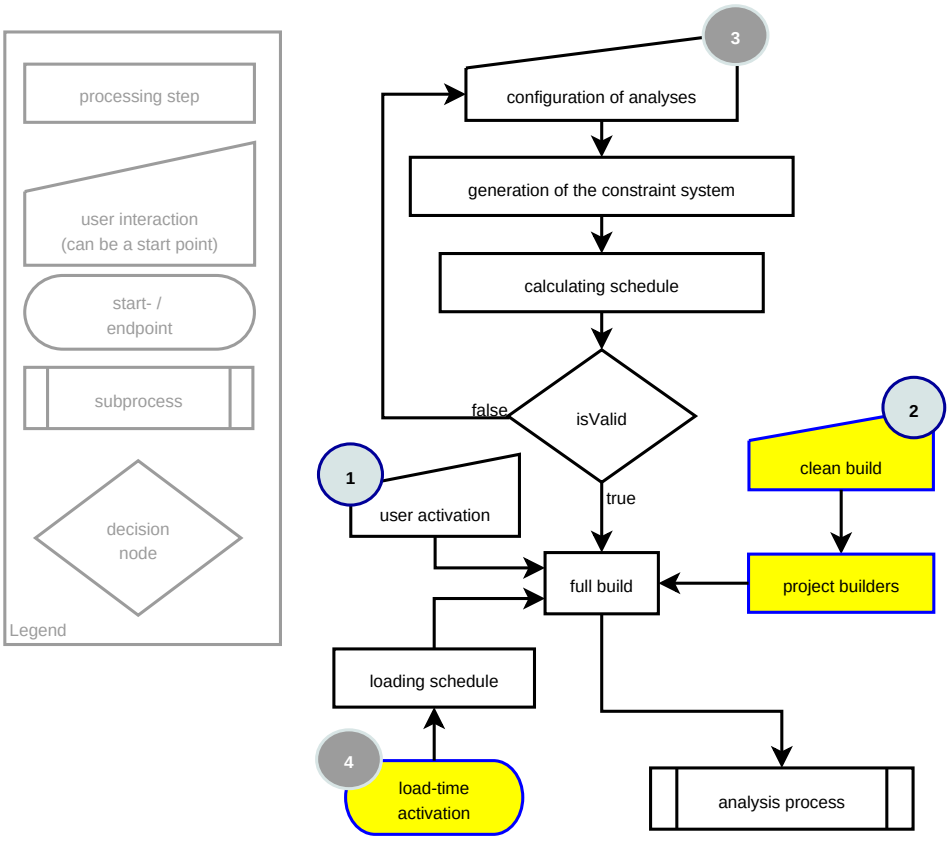

Figure 4.3: Program flows leading to full builds

Full Builds

To facilitate the comprehension of the Eclipse integration of MAGELLAN, two types of full builds are distinguished in the following discussion.

- A *clean build* is a full build triggered by Eclipse. A clean build is executed in the following two cases (marked 1 and 2 in Figure 4.3): *clean build*

  1. When the user activates the MAGELLAN plug-in for an Eclipse project MAGELLAN registers a builder to hook into the build process. During this initial build MAGELLAN only collects information about the names and locations of the project's artifacts. Since no analyses are configured yet, the schedule is empty and no analyses are executed.
  2. When explicitly requested by the user, using Eclipse's "clean build" menu item, or when a large number of the project's resources has

changed. For example, after a CVS update a clean build might be triggered.

During clean builds MAGELLAN will be invoked by Eclipse when all other project specific builders have finished. Builders are Eclipse plug-ins that process the project's resources and often generate further resources. For example, compilers and parser generators, such as ANTLR [Par06], are realized as builders. By executing MAGELLAN after all builders it is possible to analyze generated resources, e.g., Java class files.

simulated full build

- A *simulated full build* is a MAGELLAN internal full build, that is executed when all resources need to be (re)analyzed.

  A simulated full build is executed in two cases (marked 3 and 4 in Figure 4.3):

  3. When the configuration of the analyses that should be executed as part of the incremental build process changes. A full build is required to make sure that the WPDB contains the source model as derived by the resulting analysis configuration.

     When the user has finished the configuration, the constraint system is created and solved. If a schedule can be calculated, a new full build is triggered, otherwise an error is shown to the user and the user is taken back to the configuration.[1]

  4. When the Eclipse IDE is started a simulated full build is executed for projects for which MAGELLAN was previously activated. This build serves to (re)initialize the whole program databases (WPDB). Before executing the full build the last calculated schedule is restored.

Incremental Build

The program flow for incremental builds is depicted in Figure 4.4. When the user has edited and saved a project's resource, Eclipse first invokes all project builders. After that, MAGELLAN is called with the information about all resources that have changed. This includes not only user edited resources, but also all resources generated by the builders.

[1] Recall that a schedule can not be calculated if the selected analyses or analyses on which selected ones depend have conflicting requirements.

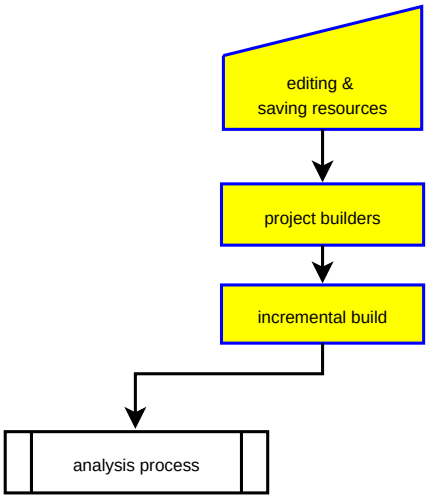

Figure 4.4: Program flow for incremental builds

Analysis Process

The input to the analysis process consists of the resources to be (re)analyzed (called *changed resources* in Figure 4.5); no distinctions concerning the build type are made. The scheduled analyses are executed in the same way whether the current build is an incremental build or a full build. However, the information about the current build type is made available to the analyses. This enables analyses to implement two different code paths: one optimized for full builds and one for incremental builds.

To execute the analyses, MAGELLAN iterates over all steps of the schedule and tries to execute the step's analyses in parallel. To parallelize the execution of the analyses MAGELLAN uses *number of processors*+1 threads. Preliminary experiments have shown that this number leads to the greatest average reduction of the overall analysis time. Static code analyses are CPU intensive once the source code is parsed; starting more threads to analyze the code is ineffective as a CPU is typically 100% utilized while executing one analysis.

Overall Program Flow

The overall program flow is presented in Figure 4.6. The figure shows how the full build, incremental build and the analysis process are related to each other. When the analysis process has finished (the $\bigoplus$ node in Figure 4.6), the user can change the configuration of the analyses, trigger a clean build,

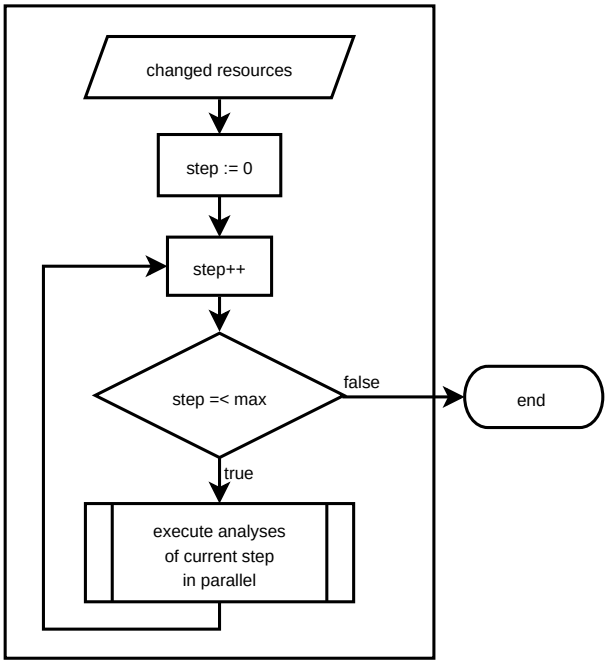

Figure 4.5: The analysis process

or can continue editing and saving the project's resources. Furthermore, on-demand analyses can be executed (the black node in Figure 4.6), e.g., to support software comprehension tools.[2]

However, after each action the project's resources are analyzed and as soon as the analysis process has finished the user can continue with the next action.

4.3 Evaluation

In the following, MAGELLAN is evaluated w.r.t. the requirements identified in Chapter 2.

Since it is a direct implementation of the approach proposed in Chapter 3, MAGELLAN fulfills the requirements: "Extensible base analyses stack"

[2]The execution of on-demand analyses that would require a different or an extended source model is currently not supported. However, supporting on-demand analyses that require a source model different to the current model is a mere engineering issue. To do so it would be necessary to: First, persist the current schedule. Second, to calculate a schedule that satisfies the requirements of the selected on-demand analysis. Third, to execute the analysis. Forth, to restore the old schedule and the corresponding whole program database.

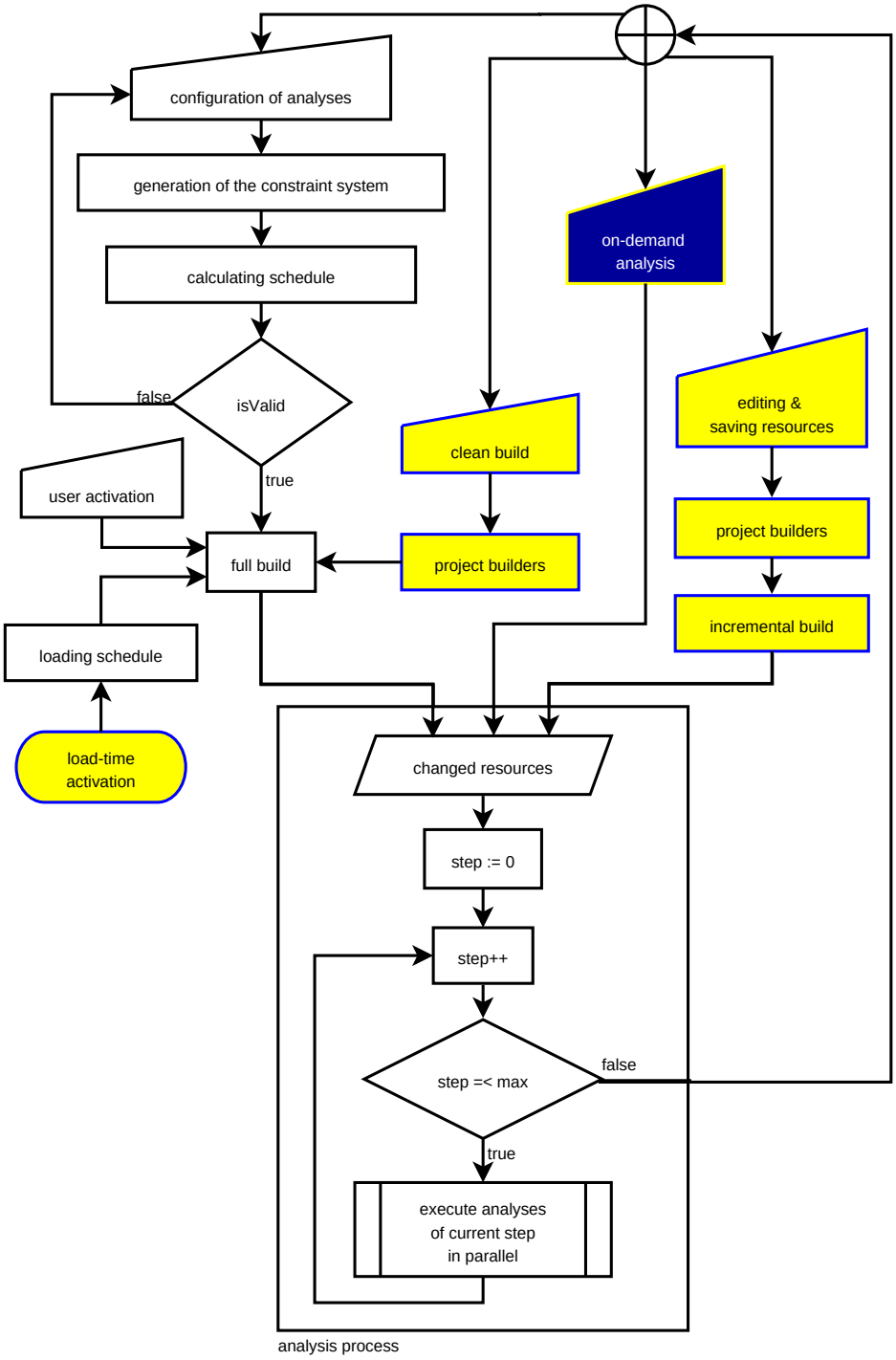

Figure 4.6: Overall program flow of MAGELLAN

(OSAP-R1), "Support for open base representations" (OSAP-R2), "Enabling cross-artifact reasoning" (OSAP-R3) and "Execution of required analyses only" (OSAP-R6).

Furthermore, being integrated with Eclipse and its incremental build process, MAGELLAN fulfills the requirement "Integrated into an IDE" (OSAP-R9); it is possible to execute analyses as part of the build process and to execute analyses on-demand. However, team support, i.e. the sharing of the configuration of the analyses, is currently not supported. The configuration is managed internally by Eclipse. This is, however, not a conceptual issue and implementing team support is just an open engineering issue.

MAGELLAN fulfills the requirement: "Support for incremental analyses" (OSAP-R7): MAGELLAN is tightly integrated with the IDE's incremental build process and detailed information about the resources that have changed since the last build are made available. Hence, the implementation and execution of incremental analyses is supported.

The requirement "Configurable set of base analyses" (OSAP-R10) is fulfilled by MAGELLAN within the limits of the approach as discussed in Section 3.5. That is, MAGELLAN enables the user to configure the set of base analyses, but context dependent scheduling of (base) analyses is not supported. E.g., a base analysis, such as a dead code analysis, will be executed after a checker (analysis) whose requirements are already satisfied before the execution of the base analysis. This is (cf. Section 3.5) not always advantageous.

Though the requirement "Support for parameterized checkers" (OSAP-R4) is supported by the MAGELLAN core, a user interface to parameterize checkers is lacking. This is not a severe restriction as a user interface could also be developed as part of a tool which builds upon MAGELLAN. The tool could then use the core's functionality related to parameterized checkers.

Meta-analyses (OSAP-R8) are not supported by MAGELLAN, i.e., it is currently not possible to use meta analyses to drive the analysis process of checkers. However, support for meta analyses can be easily implemented as described in Section 3.5.

MAGELLAN itself does not provide error report management related functionality (OSAP-R11). But, being integrated into Eclipse, analyses that require error report management can use Eclipse's problem view to show errors to the user. Unfortunately, the problem's view functionality is rather limited. Filtering of error reports is limited to filtering reports of a specific type and it is not possible to filter reports based on an error's source locations. The dynamic ranking of errors based on an error's likelihood of being a false warning is also not supported. Furthermore, error reports are limited

to a short descriptive text. Complex reports, e.g., reports containing the call chain leading to a potential dead lock, are not supported. Hence, this requirement is only partly fulfilled.

The last remaining requirement: "Enabling the embedding of query engines" (OSAP-R5) is supported by MAGELLAN as we will see in the following chapter; to fulfill this requirement no special functionality in the MAGELLAN core is required.

4.4 Conclusions

As discussed in the previous section, MAGELLAN does fulfill those requirements that are crucial for evaluating the feasibility of open static analysis platforms. That is, using MAGELLAN it is possible to execute an open set of static analyses as part of Eclipse's incremental build process. The remaining engineering issues identified in the previous section are not conceptual and are primarily related to the platform's usability. Hence, MAGELLAN enables a throughout assessment of the feasibility and scalability of open static analysis platforms.

Chapter 5

Embedding Query Engines

Part of the material in this chapter is published in: *XIRC: A Kernel for Cross-Artifact Information Engineering in Software Development Environments* [EMOS04], *Automatic Incrementalization of Prolog Based Static Analyses* [EKS+07], and in *Pointcuts as functional queries* [EMO04]

As identified by requirement OSAP-R5, enabling the integration of query engines is indispensable for open static analysis platforms. To evaluate Magellan in this regard two different query engines were integrated: an XQuery engine as well as a Prolog system.

These query languages were chosen because they are applicable for a wide range of different purposes when compared with domain specific query languages. The latter have the potential to facilitate complex analyses within a particular domain [Wuy98, HHR04, Cre97, MLL05, Vol06], but can hardly be used for different types of analyses or for analyzing code in different languages.

PQL [LL05], for example, can be used to query for security vulnerabilities based on tainted objects. In the background, sophisticated inter-procedural data-flow analyses are performed. However, analyses of structural properties or cross-artifact analyses are not in focus of PQL and are not supported. Hence, to support a wide range of analyses it would be necessary to embed a large number of special purpose query engines.

XQuery — XML's native query language — and Prolog on the other side are already used for a wide range of different purposes such as, e.g., calculating metrics, exploring software systems or implementing analyses to detect errors. Furthermore, these query languages are neutral w.r.t. the language in which the analyzed code is written.

Supporting XQuery is particularly advantageous because XML is already widely used by software engineering tools [CCS04, HSvG03, MC04, MCK04, GK02, MW04, MMFA04] and a large number of mappings between code in different programming languages and XML is readily available [CMM02, Bad00, MK00, MCM02, FvG03, MMN02, HSvGF03, HWS00, AEK05, ST03]. Prolog was chosen because it is also well known, mature, and highly optimizing Prolog engines are freely available. The query optimizations done by the Prolog systems promise to offer the performance necessary to make an integration with the incremental build process possible.

The XQuery queries are directly evaluated on top of an incrementally maintained XML view of the program representation. The Prolog queries are evaluated using a standard Prolog system. The facts are stored in an external database which is incrementally maintained.

The integration of both query engines is described in detail in the following sections. At the end of each section the embedded query engine is evaluated w.r.t. its support for the features identified as part of the requirement OSAP-R5.

5.1 Embedding an XQuery Engine

Before the integration of the query engine into Magellan is described, a short overview of XQuery is given. Only those features of XQuery are elaborated on that are necessary for understanding the examples in the following chapters.

5.1.1 Introduction to XQuery

XQuery [BCF+05] is a functional, declarative, Turing-complete [Kep04] query language designed for querying XML data sources. XQuery consists of several kinds of expressions that can be nested and composed with full generality. The most important among them is the notion of *path expressions*.[1] In a nutshell, a path expression selects nodes in an XML tree. For illustration, consider the XML document in Listing 5.1 representing the bytecode of a simple session bean class named SimpleBean (Line 1) with a default constructor: the method named <init> in Line 6.

```
1 <class name="de.tud.SimpleBean" visibility="public">
2   <inherits>
```

[1] This subset of XQuery is a separate standard called XPath [CD99].

```
3       <class name="java.lang.Object"/>
4       <interface name="javax.ejb.SessionBean"/>
5     </inherits>
6     <method name="<init>" visibility="public">
7       <signature>
8         <returns type="void"/>
9       </signature>
10      <code>
11        <load index="0" />
12        <invoke   declaringClassName="java.lang.Object"
13                  methodName="<init>">
14          <signature> <returns type="void"/> </signature>
15        </invoke>
16        <return />
17      </code>
18    </method>
19    ...
20  </class>
```

Listing 5.1: XML representation of a simple Java class file

This document can be parsed by accessing the top-level document node (class) of the corresponding tree. Then the path expression /class/method/code/invoke selects all invoke nodes, resulting in the node spanning Line 12 to Line 15 in Listing 5.1.

In general, a path expression consists of a series of steps, separated by the slash character. The previous path expression has three steps, namely the child steps method, code, and invoke. The result of each path expression is a sequence of nodes. XQuery supports different directions in navigating through a tree, called axes. In the path expression above, we have seen the child axis. Other axes that are relevant are the descendant axis (denoted by "//"), the parent axis (denoted by ".."), the ancestor axis (denoted by "ancestor::"), and the attribute axis (denoted by "@"). Using the descendants/ancestor axis rather than the child/parent axis means that one step may traverse multiple levels of the hierarchy. For example, the above query could be rewritten as: //invoke.

The attribute axis selects an attribute of the given node, whereas the parent axis selects the parent of a given node. For example, the path expression //code/../@name selects all name attributes of all method nodes that have a code child, i.e., which are not abstract methods. Another important fea-

ture of XQuery is its notion of predicates — boolean expressions, enclosed in square brackets, used to filter a sequence of values. For instance, the query //method[@name="main"] selects all methods with the name main.

One can bind query results to variables, which in XQuery are marked with the $ character, by means of a let expression, as illustrated in Listing 5.2.

```
1 let $concreteMethods := //code/..
2 return $concreteMethods/[@name = "main"]
```

Listing 5.2: A variable definition in XQuery

The `for` construct has the same syntax as let, but it iterates over all values of the sequence returned by the query.

XQuery also offers a number of operators to combine sequences of nodes, namely union, intersect, and except. The operators have the usual set-theoretic denotation, except that the result is again a sequence with a specific order.

XQuery also supports function definitions. For illustration, the function diff shown in Listing 5.3, being passed two sets $m1 and $m2 of method elements (the * in as element()* stands for "zero to many"), returns the result of the set subtraction operation applied to them.[2]

```
1 declare function diff ($m1 as element()*, $m2 as element()*)
2     as element()*
3 {
4   $m1/.. except $m2/..
5 }
```

Listing 5.3: A function definition in XQuery

The last relevant feature is that XQuery also provides XML like constructors to create XML structures within a query, as illustrated by the query shown in Listing 5.4.

```
1 <entries> for $c in //class return
2   <entry name="{$c/@name}"/>
```

[2]XQuery also has a sophisticated type system based on XML Schema [FW04]. With this type system it would be possible to make the types more specific, e.g., we could use the type method* instead of element()* for $m1 and $m2 in Listing 5.3, whereby `method` would be defined in the corresponding schema definition. This would make queries safer and more robust against programming errors and enables some XQuery engines to optimize the query execution. However, we have not used this feature as the embedded XQuery engine does not support it.

```
3 </entries>
```

Listing 5.4: XQuery where the result is a marked up XML document

The result of evaluating the query shown in Listing 5.4 for the XML document shown in Listing 5.1 is:

```
<entries> <entry name="de.tud.SimpleBean" /> </entries>
```

5.1.2 Integrating the Saxon XQuery Processor

5.1.2.1 Overview

To enable XQuery-based analyses, a pseudo-analysis was implemented that stores an object of type XMLDB in MAGELLAN's database. An instance of XMLDB manages an XML tree and provides functionality to execute XQuery queries. The class XMLDB implements the IWholeProgramFact interface (cf. Figure 3.1). The instance of the class XMLDB stored in MAGELLAN's database is referred to as the *XML database* in the following.

XML database

The XML database is populated and maintained by subsequently executed analyses which are free to store arbitrary XML data in the database — in particular XML representations of project artifacts. Tools that want to query specific information just declare a dependency on the LSV entity representing the required XML data. MAGELLAN will then execute a corresponding analysis as part of the build process to make the information available. Given the populated XML database, a query is executed by passing the XQuery to the XML database object.

In the following sections, the various aspects of the integration are explained in further detail.

5.1.2.2 The XML Database

The data stored in the database is a large object graph that represents one XML document. The object graph is constructed using the JDOM [HM06] XML library. Initially, the database only contains the root element — the db:all element shown in Listing 5.5.

The ASL file of the analysis that creates the empty database is shown next and specifies that a single LSV entity (the empty database) is maintained. Since the analysis does not analyze any project artifacts and does not have any direct or indirect dependencies on project artifacts it is called a *pseudo-analysis*.

pseudo-analysis

```
1 <db:all>
2   ...
3 </db:all>
```

Listing 5.5: The root element of the XML database

```
analysis EmbedXMLDB
  maintains XMLDB
```

A **maintains** dependency is specified to make sure that the entity is never invalidated by any other analysis, i.e., that the database represented by the LSV entity is never deleted.

In case of a full build, a new instance of XMLDB is created, in case of an incremental build no specific action is performed by the analysis.

An analysis that wants to store data in the database declares a reads dependency on the XMLDB entity to make sure that the database is available when required. Furthermore, to enable other analyses or software comprehension tools to use the information stored in the database it is necessary to specify an LSV entity that represents the added XML data. Using the declared LSV entity other analyses can then specify a dependency on the XML data. Since the data is stored in the database, the new LSV entity has to be declared as dependent on the XMLDB entity.

For example, the XML representation of Java class files created using BAT_2XML (cf. Appendix V) is represented by the entity CF_XML which directly depends on the entity XMLDB. The complete specification of BAT_2XML is as follows:

```
analysis BAT2XML
  reads Document/CF/Method/BCode/CFG, Document/CF/Field
  reads−global Document/CF
  maintains XMLDB/CF_XML
```

The first **reads** dependency states that the analysis reads the bytecode based representation (BCode) of class files. Further, the control flow graph (CFG) is also required. This information is only required for currently changed or added documents (**reads**). The second dependency (**reads−global**) specifies that information about the class's interface needs to be available also for class files that were processed during a previous build step (**reads−global**). The information about the class file interface is required by BAT_2XML

to maintain the database, in particular, to remove XML representations of outdated class files. Finally, the **maintains** dependency states that BAT$_2$XML keeps the set of XML representations of Java class files up-to-date.

The effect of executing BAT$_2$XML as part of the incremental build process is: For each class file that is added in the current build step, an XML representation is created and added to the database. For changed class files, the representation is updated and for removed class files the corresponding XML representation is also removed.

If BAT$_2$XML is executed along with the build process, the structure of the XML database will be as depicted in Listing 5.6. The two exemplary children of the db:all element represent corresponding class files and were added by BAT$_2$XML.[3]

```
1 <db:all>
2   <bat:class name="x.y.Z" visibility="public" ...>
3       ...
4       </bat:class>
5   < bat:class name="u.v.W" visibility="public" ...>
6       ...
7       </bat:class>
8 </db:all>
```

Listing 5.6: Excerpt of the XML database

5.1.2.3 Evaluating XQueries

The interface of the XMLDB class, which is the root class of the database, is shown in Listing 5.7. Additionally to enabling analyses to add and remove XML elements (Listing 5.7, Line 4–6), functionality to evaluate queries (Listing 5.7, Line 9–16) is also provided.

The query support builds upon the Saxon XQuery processor [Kay05a]. Saxon was chosen because it is a standard conforming implementation, operates completely in memory, and was implemented in Java.[4] As shown in the

[3]The namespaces (db and bat) are used to keep the database extensible.

[4]Before Saxon was chosen a large number of (XML) databases with XQuery support were evaluated. eXist[Mei05] was premature and crashed several times. Further, an update of the database took multiple seconds even for small documents. Tamino [Sof05] supported only an outdated version of XQuery with a severely restricted set of features. The same applies to Sedna [MOD05]: crucial functionality such as function definitions were not

```
1 public class XMLDB extends IWholeProgramFact {
2
3   // methods required by analyses which maintain (add / remove) facts
4   void addElement(Element element);
5   void removeElement(Element element);
6   Enumeration<Element> getElements(Namespace namespace);
7
8   // querying related methods
9   XQueryExpression compileQuery(String query, String baseURI) ...
10  SequenceIterator executeQuery(XQueryExpression exp) ...
11  SequenceIterator executeQuery(
12        XQueryExpression exp,
13        NodeInfo context)...
14  SequenceIterator executeQuery(
15        XQueryExpression exp,
16        SequenceIterator contexts) ...
17 }
```

Listing 5.7: Interface of the embedded XML database

following, being implemented in Java made it easier to provide some of the functionality identified as part of the query engines requirement (OSAP-R5).

- The executeQuery method shown in Listing 5.7 Line 10 supports queries that search the entire database, e.g., to find a type definition, or a method declaration. These queries return direct references to the nodes of the XML database that match the selection criteria. This type of query can be used to implement search features of software comprehension tools, or to search for violations of best practices and implementation restrictions.

  For example, given the XML database shown in Listing 5.5, the result of the following search query is a reference to the second child element (Listing 5.5, Line 5) of the XMLDB's root element: db:all (Listing 5.5 Line 1)

  ```
  /db:all/bat:class[@name="de.tud.SaxonWrapper"]
  ```

supported. In case of Berkely DB XML [Sle06] the evaluation time of queries was not acceptable when the indexing functionality was turned off. But, if indexing was turned on the time required to maintain the database was not acceptable.

- The executeQuery methods shown in Listing 5.7 Line 11 and Line 14 support queries that are defined with respect to a previously selected node, i.e., queries that need a specific context to be evaluated. These types of queries enable to browse through a software project.

  For example, the result of a search query that returns a node which wraps a class declaration can be set as the context for a query to get all sub- or supertypes.

The SequenceIterators returned by the executeQuery methods iterate over sequences of Nodes. A node is either some derived information, e.g., a value that represents the depth of inheritance tree for a specific class, or a Java wrapper around an element of the XML database.

Each query to be executed as part of the build process is wrapped into its own whole program analysis by mean of a Java wrapper. During an incremental build this wrapper just passes the query on to the database for evaluation and then processes the result, e.g., shows warning messages.

5.1.3 Evaluation

The features of the embedded XQuery engine are evaluated w.r.t. the OSAP-R5 requirement.

Semantic queries are supported provided that the data stored in the database is appropriately marked-up, i.e., each semantic item, such as a field's name, the modifiers or the declaring class, has to be marked-up. Given a representation with a fine-grained markup it is then possible to write queries such as: "Get all accessed fields of method X", "Get the declaring class of method Y" or "Get the superclass of Z".

Well suited representations are generated, for example, by JavaML [Bad00], srcML [MCM02] or BAT$_{\mathbf{2}}$XML (cf. Appendix V).

Query chaining is supported. If a query's result is some information directly stored in the database, e.g., an element representing a class declaration, the result can be used as the context for the evaluation of the next query, e.g., to navigate to the superclass.

Query filtering is not directly supported, i.e., no functionality is provided that can be used to filter those queries that are not applicable to a

specific database element.[5]

However, using the query chaining feature implementing query filtering is straight forward as shown in the following. Given a database element, it is possible to write a tool specific query that "types" the element. For example, given a db:all/bat:class (cf. Listing 5.5) element the query could analyze the element's path and could return the type: ClassDeclaration.

Assuming the queries declare meta-data about the supported types of elements, it is then possible to present the user only those queries that are meaningful given the current context. For the current example, to show a query to get the superclass and to omit a query that returns the declared exceptions (of a method). This approach is for example used by the SEXTANT software exploration tool (cf. Chapter 7).

Automatic incrementalization is not supported; i.e., to get the updated result of an XQuery after an incremental change the query need to be re-evaluated w.r.t. the entire database.

In short, the embedded XQuery engine directly supports semantic queries and query chaining. Though, query filtering is not directly supported, the provided functionality at least facilitates tool specific filtering of queries. Automatic incrementalization is not supported.

5.2 Embedding a Prolog System

In this section, the integration of an extended version of XSB Prolog [XSB06] in MAGELLAN is presented. Compared to the original XSB prolog engine, support for incremental tabled evaluation was integrated [SR06], i.e., the result of a query is incrementally updated when the fact base changes. The incremental evaluation feature is particularly promising to enable a tight build process integration of analyses written in Prolog.

First, an overview of implementing static analyses using Prolog is provided before the automatic incrementalization of analyses is discussed. The integration of XSB in MAGELLAN is presented afterwards.

[5] Using Saxon, it is possible to evaluate every XQuery w.r.t. every possible element of the XML database. However, depending on the query the result set might be empty or even worse contain unexpected elements.

5.2.1 Writing Analyses using Prolog

Two example analyses are presented to illustrate the approach to specifying static analyses as Prolog queries. The first example analyzes a class' interface to detect violations of a best practice in applying the visitor design pattern [GHJV95]. The second example performs an intra-procedural data-flow analysis to control the creation of aliases.

Example I

When implementing the visitor design pattern it is a best practice to implement a special visit method for each type in the visited hierarchy.

For illustration, consider the Java code in Listing 5.8. The classes Node (Line 3) and StructureVisitor (Line 12) are defined together at some point in time. Later on, the class SubNode (Line 7) is added to the code base. This, however, violates the best practice mentioned above: StructureVisitor does not implement a visit method for SubNode. Nevertheless, the compiler will not generate any warning. In the following, a Prolog-based static analysis for detecting such a violation is presented.

```
1 package bat;
2
3   public class Node{
4     void accept(Visitor visitor){visitor.visit(this);}
5   }
6
7   public class SubNode extends Node{
8     /* empty */
9   }
10
11  @Visitor(Node.class)
12  public class StructureVisitor{
13    public void visit(Node node){...}
14  }
```

Listing 5.8: Sample implementation of the Visitor design pattern

Listing 5.9 shows the Prolog encoding of the source code. A class fact (Line 5, 8, or 12) consists of the package, the fully-qualified class name, the visibility, boolean values denoting whether the class is final or abstract, and the superclass. The first value in method facts (e.g., 4 in Line 6) is a generated unique identifier for a method; after that, the declaring class is

```
1 %%class(PackageName,ClassName,AccessSpecifier,IsAbstract,IsFinal,ParentClass)
2 %%classAnn(Class,Annotation)
3 %%method(Id,DeclaringClassName,Name,AccessSpecifier,...,ReturnType,
      ListofParam,ListofAnnotations)
4
5 class('bat',ref('bat.Node'),public,false,false,ref('java.lang.Object')).
6 method(4,ref('bat.Node'),'accept',default,...,void,[parameter(ref('bat.Visitor'),[])
      ],[]).
7
8 class('bat',ref('bat.StructureVisitor'),public,false,false,ref('java.lang.Object')).
9 classAnn(ref('bat.StructureVisitor'),annotation(type('Visitor'),value(ref('bat.Node'
      )))).
10 method(2,ref('bat.StructureVisitor'),'visit',public,...,void,[parameter(ref('bat.Node
      '),[])],[]).
11
12 class('bat',ref('bat.SubNode'),public,false,false,ref('bat.Node')).
```

Listing 5.9: Encoding of source code as Prolog database

specified, followed by the method's name, its visibility (default is the assumed visibility in Java when no visibility is explicitly specified), an encoding of the method's modifiers using boolean values (omitted for brevity), the return type, the parameter types along with parameter annotations and the list of declared exceptions.[6]

The analysis is specified as the visitor(Class) query in Listing 5.10 Line 13. The query identifies visitor classes declared as such via the @Visitor(Type) annotation that do *not* implement a visit method for every subtype of the annotation parameter: For doing so, the query first selects classes with the @Visitor annotation to get the root of the visited hierarchy: Node in our example. Next, it applies the rule transinvinherits/2 to find all classes which extend Node; for any such class, the query verifies that the Visitor has a corresponding visit method and if not, the class is bound to the variable Class. As the result of evaluating the query, warnings are generated for each answer to the query, i.e., for each binding of the variable Class. Each such class violates the best practice of the visitor design pattern.

Example II

[6] In Prolog, angular brackets [...] are list constructors; variables start with an uppercase; the special character _ denotes anonymous variables.

```
1 % the subtype relation is computed by invinherits and transinvinherits
2 invinherits(Interface,Class):- classInterfaces(Class,Interface).
3 invinherits(ParentClass,Class):- class(_,Class,_,_,_,ParentClass).
4 invinherits(X,Y):- interfaces(Y,X).
5
6 :- table transinvinherits/2.
7 % transitive reflexive hull of invinherits
8 transinvinherits(X,Y) :- invinherits(X,Y).
9 transinvinherits(X,X).
10 transinvinherits(X,Y) :- invinherits(X,Z), transinvinherits(Z,Y).
11
12 :- table visitor/1.
13 visitor(Class):- classAnn(Visitor,annotation(type('Visitor'),value(Node))),
14                   transinvinherits(Node,Class),
15                   not(hasmethod(Visitor,'visit',[Class])).
```

Listing 5.10: Query to check implementations of the Visitor design pattern

This analysis checks that a method does not return the self reference (this). Such a check is, e.g., required when implementing confined types [VB01]. This analysis is based upon the prolog encoding of the 3-address based code representation [Sco00] in static single assignment form [CFR+91] that is provided by BAT [BAT06]. In this representation data-flow information is made explicit and, thus, implementing data flow analyses is simplified.

A violation of the constraint that this is never returned is shown in Line 5 of Listing 5.11: this is assigned to the variable o which may be returned by the method later on without being assigned a new value in-between.

```
1 /*@Confined*/ class C {
2   public Object violate(){
3     Object o;
4     if (...)
5       o = this;
6     else
7       o = null;
8     return o;
9 } }
```

Listing 5.11: The self reference this may be returned (Java)

```
1 method(4,ref('C'),'violate',public,...).
2   if(4,2,4,...,operator,...,1).
3   label(4,3,4).
4   goto(4,4,4,2).
5   label(4,5,1).
6   label(4,7,2).
7   phi(4,8,8,p7,[phiElem(thisValue,4),phiElem(nullType,1)]).
8   return(4,9,8,p7).
```

Listing 5.12: The self reference this may be returned (Prolog encoding)

Listing 5.12 shows the prolog encoding of the method shown in Listing 5.11. The first value of each method implementation fact (Line 2–8) is the id of the method (Line 1, first value) and the second one is the number of the instruction. The third value is the line number of the corresponding source code, except for labels (Line 3,5,6) where the value is a method-wide unique id. The last values of if (Line 2) and goto (Line 4) statements are the id's of labels that are the jump targets. Besides being used as the targets of jump instructions, labels are also defined for each basic block of the control flow graph. The phi statement (Line 7) is a result of the transformation into static single assignment form and states that the value of the (helper) variable p7 (line 7) is control flow dependent: If the id of the basic block of the last executed instruction is 4 the value of p7 will be this (phiElem(thisValue,4)). If the basic block's id is 1 the value will be null.

The query to detect the violation is shown in Listing 5.13. initializedWithThis (Line 1–3) is a helper predicate that binds its second argument to variables directly initialized with thisValue or to thisValue itself. The analysis is defined in Line 5 – 7. Line 6 binds RetVal to variables that are directly or indirectly initialized with thisValue. Line 7 succeeds for those methods that return such a value.

```
1 initializedWithThis(MethodID, Variable) :-
2     def(MethodID,_,_,_,Variable,thisValue) |
3     (phi(MethodID,_,_,Variable,Phis) , member(phiElem(this,_),Phis) ).
4 initializedWithThis(_, thisValue).
5
6 returnsThis(MethodID) :-
7     initializedWithThis(MethodID, Val), propagate(Val, RetVal),
8     return(MethodID,_,_,RetVal).
```

Listing 5.13: Prolog based analysis to detect methods that return this

The predicate propagate/2 (Listing 5.14) is the reflexive and transitive closure of all initializations of a variable. dpropagate (Line 1) implements the initialization relation.

```
1 dpropagate(V1, V2) :- phi(_,_,_,V2,Phis), member(phiElem(V1,_), Phis).
2 propagate(V,V).
3 propagate(V1,V2) :- dpropagate(V1,V2).
4 propagate(V1,V2) :- dpropagate(V1,V3), propagate(V3,V2).
```

Listing 5.14: The reflexive and transitive closure of all variable initializations

As shown by the propagate predicate, analyzing the data-flow is simplified as each variable is initialized exactly once and the data-flow is explicitly encoded in the phi facts.

To further illustrate the advantage of the chosen code representation consider the code shown in Listing 5.15. In this case, the constraint that this is never returned is not violated as the initialization of o with this (Line 2) will never reach the return statement (Line 8). Since the use-def chains are explicitly encoded, an analysis of return o (Line 8) immediately reveals that o is always assigned a new instance of an Object (Line 9) and that the assignment of this in Line 2 is not relevant. Hence, an analysis of a method's control flow graph is superfluous.

```
1 public Object noViolation(){
2   Object o = this;
3   try {
4     do something ;
5   } finally {
6     o = new Object();
7   }
8   return o;
9 }
```

Listing 5.15: The self reference this is not returned

5.2.2 Automatic Incrementalization of Analyses

Since tabled evaluation of Prolog programs is the foundation for the automatic incrementalization, it is explained first.

5.2.2.1 Tabled Evaluation

Tabled logic programs declare certain predicates as tabled. Recursive predicates (for ensuring termination) and predicates that are re-used multiple times are good candidates for tabled predicates. Tabled resolution systems evaluate programs by memoizing subgoals of tabled predicates (referred to as *calls*) and their provable instances (referred to as *answers*) in a set of tables.

Calls are stored in a call table and all answers corresponding to a call are stored in a corresponding answer table. During resolution, if a subgoal is present in the call table, then it is resolved against the answers recorded in the corresponding answer table (*answer clause resolution*); otherwise, the subgoal is entered in the call table, its answers are computed by resolving the subgoal against program clauses (*program clause resolution*), and are entered in the answer table.

The principles of tabling are exemplified using the visitor example. As shown in Listing 5.10 Line 6, the recursive predicate transinvinherits/2 is declared as tabled. Also the top-level predicate visitor/1 is declared as tabled (Line 12); a query visitor(Class) can be resolved by looking up the visitor(Class)'s answer table if the latter is non-empty. When visitor(Class) is executed for the first time, tabling creates an entry visitor(Class) in the call table and uses the rule for the visitor predicate to find answers.

Resolving the first subgoal of the visitor predicate binds the variables Node and Visitor to ref('bat.Node') and ref('bat.StructureVisitor') respectively. The transinvinherits predicate is evaluated with the call transinvinherits(ref('bat.Node'),Class), which is stored in the call table. The answers Class=ref('bat.Node') and Class=ref('bat.Subnode') of this call are obtained by resolution of the second clause of transinvinherits, and by resolution of the first clause of transinvinherits and invinherits, respectively. These answers are stored in the answer table of the transinvinherits(ref('bat.Node'),Class) call. The resolution of the last subgoal in the body of the visitor predicate generates only the answer Class=ref('bat.Subnode') for the call visitor(Class), as the last subgoal fails for the substitution Class=ref('bat.Node'). Since visitor/1 is tabled any subsequent visitor(X) call will be resolved from its answer table.

5.2.2.2 Incremental Evaluation

Any change to the Java program causes the addition and deletion of facts to the Prolog fact base. Changes in the fact base can, in turn, render already evaluated tables stale: They may not have all the answers or the answers in the tables may be incorrect. The *non-incremental* approach to this problem

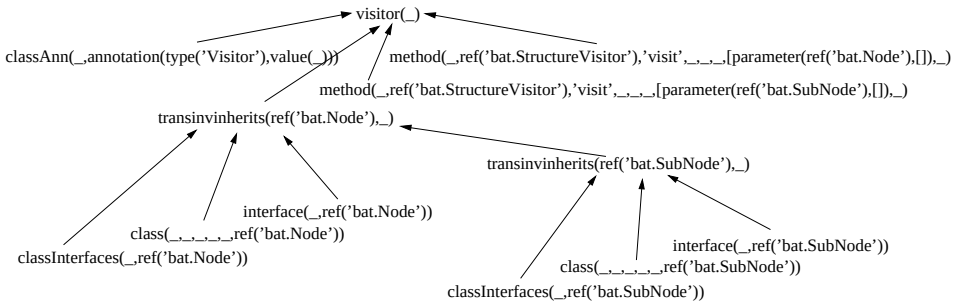

Figure 5.1: Call graph for Visitor example

is to abolish all the call and answer tables, and reissue the query. This is often wasteful, specially when the effect of the changes to the fact base is small. On the contrary, the incremental evaluation algorithm tries to identify the calls that are *changed* and re-issues only these calls. The algorithm is presented in [SR06] and is shortly described in the following.

A call is deemed changed iff the set of answers corresponding to the call before the change differs from that after the change. However, it is not possible to identify the set of changed calls before re-evaluating any calls. Thus the incremental algorithm *over-approximates* the set of changed calls by the set of *affected calls*, which are calls that can be potentially changed.

To determine the set of affected calls, the incremental algorithm maintains a data structure which keeps the dependency between calls and facts that can be changed (known as *volatile* facts). The data structure, known as *called-by graph*, is central to the incremental algorithm and is described below using the visitor example.

Informally, the called-by graph is a directed graph whose nodes consist of calls and subgoals that unify with the volatile predicates. A path from a node c_1 to node c_2 indicates that c_1 is a tabled subgoal (or a call to a volatile predicate) that was called while resolving the tabled subgoal c_2. Each edge describes the immediate dependency between calls. The graph captures the dependencies between tabled calls and calls to volatile predicates. It is first generated in the initial (non-incremental) run, and maintained over subsequent incremental runs.

The called-by graph for visitor(Class) is given in Figure 5.1. The edges from the nodes clsAnn(_,annotation(type('Visitor'),value(_))), transinvinherits(ref('bat.Node'),_), and the two method nodes to node visitor(_) correspond to the first, second and two calls to the third subgoal in the body of clause visitor(Class), respectively.

The incremental algorithm works in two phases: an *invalidation* phase and a *re-evaluation* phase. The invalidation phase finds affected calls by bottom-up traversing the called-by graph starting from the vertices that unify with added or deleted facts. Edges in the called-by graph are directed from callee to caller which enables to compute the affected calls by traversing the called-by graph. For an illustration, consider the addition of a StructureVisitor.visit(bat.SubNode) method. This adds a fact similar to the one in Line 10 of Listing 5.9, which instead of bat.Node refers to bat.SubNode. The invalidation phase determines the visitor(_) call as affected, because the added fact unifies with the method node of the called-by graph that has ref('bat.SubNode') as a parameter, which, in turn, has a path to node visitor(_).

If an added/deleted fact does not unify with any leaf of the called-by graph, none of the calls are affected, i.e., the change has no effect to the present set of calls and answers. For example, if we add a class bat.Foo which does not affect the class hierarchy of bat.Node, none of the existing leafs will unify with the added class fact for bat.Foo. Hence, none of the existing calls are affected and re-evaluated. On the contrary, a non-incremental evaluation will re-evaluate all existing calls.

5.2.3 Integrating XSB Prolog

The integration of Prolog into MAGELLAN is comparable to the integration of the XQuery engine and, therefore, only briefly described in the following.

As with the embedded XQuery engine, a pseudo-analysis stores a whole program fact — PrologDB — in MAGELLAN's database. In case of a full build a new instance of the database is created; during incremental builds no special action is performed. The PrologDB fact provides the interface to interact with the Prolog database[7] including methods to:

- assert and retract facts (e.g. as shown in Listing 5.12, 5.13)
- consult prolog rules (e.g., to consult the file with the definitions of the Prolog rules shown in Listing 5.14)
- evaluate queries (e.g., visitor(Class))

[7]Unlike the XQuery engine, the Prolog system is executed in an external process and the communication between MAGELLAN and Prolog is based on exchanging messages using TCP/IP.

The database is populated by subsequently executed analyses. Analyses that store data in the database have to declare a dependency on the database and have to specify LSV entities that represent the added data.

For example, the prototypical analysis that transforms BAT's 3-address based representation of Java class files into a Prolog encoding (cf. Section 5.2.1) has the following specification:

```
analysis BAT2Prolog
  reads Document/CF/Method/QCode, Document/CF/Field
  reads−global Document/CF
  maintains PrologDB/CF_pl
```

Using the assert and retract functionality the analysis maintains the set of Prolog representations of the project's class files represented by the LSV entity CF_pl (Line 4 of the above listing).

Furthermore, common rules used by other queries, e.g., transinvinherits in Listing 5.10 and propagate in Listing 5.14, are made available for subsequent analyses by using (pseudo-)analyses that consult the rules in case of a full build. To consult rules means to load a prolog file with a set of (commonly used) rule definitions. After loading, the rules can be used by other queries or rule definitions. Consulting rules only when required is necessary to avoid the maintenance of tables associated with rules that no user selected analysis uses. For example, maintaining the table for transinvinherits (Listing 5.10, Line 6) would waste memory and processing time, if transinvinherits is not required by any subsequently executed query.

The analysis that consults the rules for propagate and transinvinherits has the following specification:

```
analysis BAT2CommonRules
  maintains PrologDB/propagate, PrologDB/transinvinherits
```

Analyses that want to make use of the predefined rules just have to declare a dependency on the LSV entities (e.g. propagate) produced by corresponding pseudo-analyses.

Each query is wrapped by a small Java class that is called by Magellan during the build process. When invoked, the Java class passes the query to the Prolog system for evaluation and then processes the result, e.g., shows warning messages.

5.2.4 Evaluation

The features of the embedded Prolog system are evaluated w.r.t. the OSAP-R5 requirement.

Semantic queries are supported. Data stored in the Prolog database is well structured and enables a distinction between different semantic items, such as, the name of a class, a method or a field.

Query chaining is not directly supported. The result of the evaluation of a Prolog query is a set of String objects without any further meta-data about the kind of information encoded by them. It is even not possible to distinguish between derived information, such as the depth of the inheritance tree and information related to facts stored in the database, such as the name of a class. Hence, a query's result cannot directly be used as the context for evaluating another query.

Query filtering is not directly supported. Since the result of a query is a plain sequence of characters, no context information is directly available that could be used to decide which other queries are applicable.

Automatic incrementalization is supported, i.e., to use automatic incrementalization for queries it is sufficient to mark them as tabled. After that, the query's results are maintained incrementally whenever the fact base changes. On a change, only the subset of the database is (re)analyzed that is necessary to update the query's results.

The Prolog system's support for automatic incrementalization is promising to enable the integration of analyses with the incremental build process. However, due to the lack of direct support for query chaining and query filtering the Prolog system is less well suited as a back-end for software exploration tools. More effort would be required to implement exploration functionality when compared to using the embedded XQuery engine.

5.3 Conclusions

As discussed in this chapter, MAGELLAN facilitates the embedding of in-process query engines (XQuery) as well as query engines that are executed in an external process (Prolog System). MAGELLAN's architecture has proven to be flexible enough to enable the seamless integration of query engines.

Since the embedding of the query engines did not require any explicit support in the core, other query engines can be integrated when needed.

Further, no special functionality need to be implemented to support the evaluation of queries on demand of the user or as part of the incremental build process. Whether an analysis is triggered by the user or automatically by Eclipse is fully transparent for the query engine. However, since the Prolog system supports the automatic incrementalization of static analyses, it is potentially better suited to enable the execution of queries along the build process. The XQuery engine can not automatically incrementalize queries, but directly supports query chaining, which is not supported by the Prolog system. Hence, the XQuery engine is potentially better suited as the back-end of software comprehension tools that provide means to navigate / explore the project.

To sum up: Currently, none of the embedded query engines supports all functionality identified as part of requirement OSAP-R5. But, taken together all features are met.

Nevertheless, open questions remain:

- What are the performance (memory and analysis time) characteristics when using the query engines? Which size of projects can be analyzed?
- How much faster are automatically incrementalized queries when compared with queries that are not incrementalized? Is the performance of automatically incrementalized queries sufficient to enable the simultaneous evaluation of several queries along with the build process?
- Is the assumption that the embedded XQuery engine is better suited for software comprehension tools and that the Prolog system is better suited for build process integrated static analyses correct?

These questions are answered in the following chapters where applications of Magellan are presented that make use of the query engines.

Part III

Applications of Magellan

Chapter 6

Lightweight Static Analyses

This chapter shares some material with: *Using Annotations to Check Structural Properties of Classes* [ESM05] and *Enforcing System-Wide Properties* [EMS+04]

6.1 Introduction

Static program analysis is becoming increasingly used to detect problems before software is deployed. Traditionally, problems that may occur across application domain and project boundaries have been the target of static analysis, e.g., array index out of bounds, null-pointer dereferences, buffer overflows or memory leaks. More recently, the target is moving toward domain and project specific problems. Examples are problems related to multi-threaded applications [EA03, AB01], to Web and EJB applications [RSS+04a, RSS+04b, Liv04], to the usage of specific APIs [Liv05, BR02], to the violation of security constrains [MLL05], or to detecting language specific bug patterns [HP04].

Unfortunately, the built-in support of current IDEs for error detection is limited to the possibilities offered by syntax checking and type checking. Examples of checks that modern IDEs, such as Eclipse [Ecl06] and NetBeans [Net06], can perform beyond syntax checking or type checking are: the detection of the bug pattern "accidental boolean assignment"; i.e., when a developer writes `if (a = b)...` instead of `if (a == b)...`, and a check for the best practice that "`Serializable` classes should define a field `serialVersionUID`". Other analyses, such as, null-pointer dereference checks and array bounds checks [JS99], let alone domain specific checks are not supported.

The limitations of the built-in bug checking capabilities of IDEs have led

to the development of a multitude of plug-ins by third parties [EMS+04, Har05, HP04, Liv04, Liv05, RSS+04a, TAT06] to check for violations of implementation restrictions or for common bug patterns.[1] However, with respect to these plug-ins the following problems can be identified:

No Tight Integration

Some of the plug-ins are not tightly integrated into the IDEs [Liv05, RSS+04a]; only the tool's user interface (UI) is integrated, but there is no integration of these tools with the incremental build process. Hence, these tools must explicitly be started and will then perform the analysis as if they were invoked from the command line.

If analyses are not integrated into the incremental build and analysis process it is not possible to provide immediate feedback in case of an error. But, immediate feedback is of particular importance in case of cascading errors when a small (accidental) change, e.g., to the type hierarchy, causes dozens of errors to appear. Without immediate feedback, the developers will continue editing the source code. It is only after the next build of the project, that they are confronted with dozens of errors. Having to figure out which change caused the error messages and which error message is the relevant one will certainly take a larger fraction of time, when compared to using an IDE with immediate feedback.

Limited or No Extensibility

Most tools are tailored for one specific application domain and are only extensible with respect to this particular domain, if at all. E.g., Saber [RSS+04a] analyzes J2EE projects to detect method calls that do not adhere to the protocol specified in the J2EE specification and FindBugs [HP04] detects bug patterns in Java projects. Since the source model of these tools is fixed, it is hardly possible to extend them to detect other kinds of errors and or violations of specifications.

Hence, it would be necessary to install a multitude of different tools to be able to detect a wide range of bugs. This increases the complexity of the development process since it is unreasonable to expect from a developer to use a multitude of different tools for a similar purpose.

[1] In case of IntelliJ's IDEA IDE [Int06a] a former plug-in for static code analysis is now integrated with the platform.

In the following, MAGELLAN is used as a foundation for the development of analyses that are integrated into the build process. This enables an assessment of MAGELLAN's potential for eliminating the need for different static analysis tools.

The definitions of the terms: "checker" and "base analysis", which are used in the following, are shortly repeated here for the reader's convenience (cf. Chapter 1). Analyses that check that a specific property holds, e.g. that the return value of the `String.concat(...)` method is not ignored, are called *checkers*. All other analyses that derive (intermediate) information required by checkers are called *base analyses*. The term *analysis* is used to refer to checkers and base analyses.

This chapter is structured as follows. In Section 6.2 the implementation of checkers using the Bytecode Analysis Toolkit (BAT) is discussed. Section 6.3 discusses the implementation of analyses using XQuery. Section 6.4 concludes this chapter.

6.2 Checking Code using the Bytecode Analysis Toolkit (BAT)

The Bytecode Analysis Toolkit (BAT) is a library explicitly targeting the implementation of static analyses. BAT facilitates intra-procedural control- and data-flow analyses by providing a 3-address based representation [Sco00] of Java bytecode in static single assignment form [CFR+91]. This representation is heap-based and each local variable is initialized exactly once. Further, the use sites of a local variable are also made available, i.e., the local variables' definition-use chains are made explicit. Taken together these features significantly ease the development of checkers that require data flow information — as we will see in the following. Furthermore, to improve the memory footprint and to improve the performance and accuracy of analyses using BAT's representation, intra-procedural constant propagation and dead code elimination is also performed. This representation is referred to as the *quadruples representation* [Sco00] in the following.[2]

quadruples representation

To foster comprehension of the quadruples representation, a Java method and its quadruples representation are depicted side-by-side in Figure 6.1. The

[2]Higher level representations have many more applications than finding programming errors, e.g., the representation called *Jimple*, generated by the Soot framework [VRGH+00], is used by the Bandera tool suite [CDH+00] in the context of model checking. However, discussing these applications is not in the scope of this thesis.

Java method is shown on the left and the quadruples representation is shown on the right.

```
1 int doIt(){                    int doIt(){
2   int i = 5;
3   System.out.print(4);           java.io.PrintStream p1 = java.lang.System.out
4                                  /*java.io.PrintStream*/p1.print(4)
5   if(...) ...;                   if(...) ...
6   factorial(i);                  /*int p2 =*/Math.factorial(5)
7   return i;                      return 5
8 }                              }
```

Java Representation | Quadruples Representation

Figure 6.1: A Java method and its quadruples representation

Given the quadruples representation it is, e.g., immediately evident that the value "5" is passed to the method factorial (Line 6) and that the return value is constant and is also "5" (Line 7). No data-flow analyses are necessary. Furthermore, compound statements (expressions) are split up into sequences of primitive statements (expressions) to facilitate code traversal. For example, the statement out.print(4) is split into two primitive statements:

1. The statement that reads java.lang.System's out field and assigns it to the local variable p1 (Line 3).

2. The invocation of the method print (Line 4).

Further, using the definition-use information it is, e.g., trivial to detect calls to methods where the return value is ignored; ignoring the return value is an error in many cases (cf. Chapter V). The analysis that detects that the return value of the factorial function is ignored is shown in Listing 6.1.

```
1 checkMethod(...,QuadruplesCode code,...) {
2   Quadruple q = code.getFirstQuadruple();
3   while (q != null) {
4     if (q instanceof Def) {
5       Def def = (Def) q;
6       if (def.getExpression() instanceof InvokeFunction) {
7         if (def.getUseSites().length == 0) {
8           // generate error: the return value of a method call is ignored
9     } } }
10    q = q.getNextQuadruple();
```

```
11 } }
```

Listing 6.1: Return value is ignored

The analysis iterates over the linked-list of all statements (quadruples) of a method's implementation (Lines 2, 3 and 10). An error is reported (Line 8) for all local variables (Line 4) that have no use sites (Line 7) and that were initialized by the return value of a method (Line 6).

Finally, to illustrate the advantage of the static single assignment form consider the code shown in the following example.

```
1 Object violate(){          Object violate(){
2   Object o;                  boolean p1 = VALUE
3   if (VALUE)                 if(p1 == 0) goto 5
4     o = this;                goto 6
5   else                       /* nop */
6     o = null;                p2 = φ(this [←4],null [←5])
7   return o;                  return p2
8 }                          }
```

Now, let's assume that we want to determine if the self reference this is returned by the method, e.g., to make sure that no aliases are generated. Using the quadruples representation this analysis is particularly easy to implement. It is sufficient to navigate from return statements (e.g., as shown in Line 7) to the definition of the returned local variables (e.g., p2 in Line 6) and to check if the variables are initialized with this. In this case, p2 is initialized by a so-called phi statement (Line 6), meaning that the value of p2 is control-flow dependent: p2 is this if the phi statement is reached coming from Line 4 and null if coming from Line 5. However, the information which value is returned in which case can be ignored, as it is sufficient to check that p2 is not initialized with this. Hence, without any further control- / data-flow analysis it can be concluded that this might be returned.

6.2.1 Implemented Analyses

Overview

To evaluate MAGELLAN as a platform for developing and executing lightweight static analyses, 5 checkers that analyze only a class's interface and 25 different checkers that perform intra-procedural analyses were developed (cf. Appendix V).

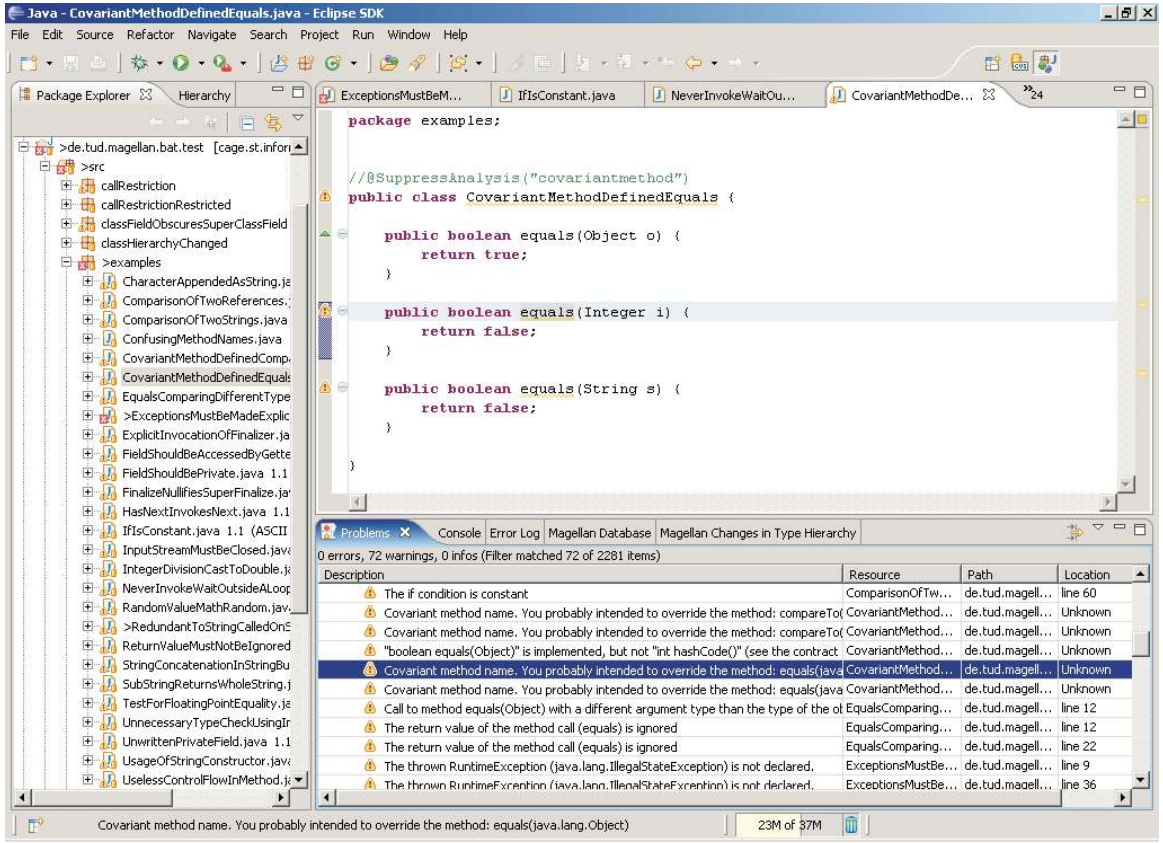

Figure 6.2: Eclipse showing checker generated error reports

A screenshot with errors reported by the checkers is shown in Figure 6.2. The errors are listed in the problems view at the bottom. By clicking on an error report it is possible to navigate to the corresponding source code. Hence, from an end-user perspective it is not distinguishable if an error report is generated by the Eclipse compiler or by one of the additional checkers.

The type of violations detected by the checkers are described in detail in Appendix V. A more detailed discussion of the implementation of the analyses is omitted since all checkers use the same representation and are widely comparable.

Embedding into Magellan

All checkers are Java classes that implement Magellan's interface for analyses. This interface defines a run method that is called by Magellan (cf. Section 4.2) to start the analysis. When calling the run method Magellan passes the WPDB to the analysis. To specify the analysis' dependencies the class is annotated using the @ASL annotation (cf. Section 4.1) which specifies the location of the ASL file.

The prototypical ASL file of the 5 class-interface related checkers is shown next. It specifies that method signatures and field declarations of class files are read (analyzed).

```
reads Document/CF/Method
reads Document/CF/Field
```

The checkers that analyze method implementations either operate on the bytecode based representation (e.g., the checker which reports redundant calls to the toString method of String objects), or use the quadruples code representation if data-flow information is required (e.g., the checker to ensure that the InputStream must be closed).

The ASL files of checkers in this category are basically the same except that some specify to read the BCode entity while the others specify to read the QCode entity, as shown in the next example.

```
reads Document/CF/Method/QCode
```

6.2.2 Performance Evaluation

To assess the suitability of Magellan for executing static analyses along with the incremental build process, a comprehensive performance study was carried out.

For this performance evaluation all 30 checkers were executed as part of the incremental build process. Including base analyses, 47 analyses were executed altogether. The times required by the checkers — including the time required by the base analyses — were measured while editing the BAT project. The BAT library consists of approximately 800 classes.

The time required by the analysis was measured while editing the source code as described in Table 6.1. These use cases resemble typical actions of developers when evolving and maintaining a software system. The use cases were chosen to cover:[3]

- changes that directly affect only a single source file, e.g., if a comment is changed.
- changes where only one source file is changed, but which requires to reanalyze multiple resources, e.g., when the type hierarchy changes.
- changes to multiple source files, e.g., when the developer performs "rename method" refactorings.

To help understand the effect of an action on the project's resources the third column of the table lists the number of resources that have changed. This includes the changed source files (e.g., the ".java" files) as well as generated files (e.g., the ".class" files) that were updated or created.

For example, if a method body is commented out, it's obvious that the Java source file changes. However, the class file also changes as the source file is immediately recompiled by Eclipse's Java compiler. Furthermore, if the class that is recompiled also includes inner class definitions, multiple class files are updated, since each inner class is compiled into its own ".class" file. Hence, editing a single Java source file might affect a larger number of resources.

The performance evaluation was made on a 3 GHz Dual Core Pentium D with 2 GB of RAM running Windows XP, Java 5 and Eclipse 3.2 RC4. To assess the effect of the automatic parallelization on the overall analysis time the analyses were executed twice: once with both cores of the CUP enabled (column 5 of Table 6.1) and once with only one core enabled (column 4 of Table 6.1). The relative performance gain when using two cores is shown in column six.

[3]Recall that in MAGELLAN the unit of change is whole document. Hence, the entire document is reported as changed when, e.g., a field is added or a method is removed.

	Description	added / removed facts	msecs. one CPU	msecs. two CPUs	%
1	*full build after Magellan startup*	1624/0	9077	6847	24.57
2	the method body of a method which declares to return a value is commented out	6/6	162	120	25.93
3	the previous change is undone	6/6	99	88	11.11
4	the method body is commented out again	6/6	97	84	13.40
5	the previous change is undone	6/6	98	89	9.18
6	an interface used by more than 250 classes is renamed	540/540	3818	2946	22.78
7	previous change is undone	540/540	3845	3002	21.92
8	a method is added to a interface	6/6	1030	391	62.04
9	the previous change is undone	6/6	792	409	48.36
10	a new package is added	0/0	14	18	-28.57
11	a new class that extends Exception is created	2/0	138	17	87.68
12	a method that overrides another method is implemented	2/2	19	14	26.32
13	a new class is created	2/2	161	16	90.06
14	a method is added to the newly created class	2/2	17	14	17.65
... continued ...					

	Description	added / removed facts	msecs. one CPU	msecs. two CPUs	%
15	the type hierarchy of the class which extends Exception is changed (extend RuntimeException)	2/2	115	87	24.35
16	a method's throws declaration is removed	2/2	24	13	45.83
17	the superclass of the newly created class is changed to (again) Exception	3/3	120	90	25.00
18	the superclass of the class created in step 11 is changed to Throwable	2/2	17	13	23.56
19	the superclass of the class changed in the previous step is changed to RuntimeException	3/3	116	89	23.28
20	the superclass of the class changed in the previous step is changed to IllegalStateException	2/2	15	13	13.33
21	the complete package created in step 10 is deleted	0/4	126	123	2.38
				average benefit	28.10

Table 6.1: Performance figures of BAT based analyses

As the performance figures show, the time required by the analyses during an incremental build is always less than one second on a dual core CPU system. Even if only a single core is available the execution time is at most one second. The change performed in step 6 represents a major refactoring that affects approximately one third of all classes of the project. Hence, this action represents an editing action that is not executed frequently and for

which developers are likely to accept longer build and analysis times.

Furthermore, the figures show that the automatic parallelization is effective. On a dual core system the overall execution time of the analyses is in average 28% shorter when compared to a single core system.

6.3 Checking Structural Properties using XQuery

In the following, the use of XQuery for checking structural properties is evaluated.[4]

The focus is on using (Java) annotations as hooks for binding the analyses. Annotations are becoming more and more widespread and already do change the way enterprise applications are built. All major Java standards such as EJB 3.0 [EJB05], Java Web Services [Web05], or JDBC 4.0 [JDB06] (will) use annotations. Further, a set of annotations is currently under development that applies across a variety of individual J2SE and J2EE technologies [Ann06]. It is expected that the use of annotations will make the development process of components more lightweight and will flatten the learning curve of the technologies.

A Java annotation is a modifier, such as public, static and final, that can be used as part of package, type, constructor, method, field, parameter, and local variable declarations. An annotation has a type and defines zero or more member-value pairs, each of which associates a value with a different member of the annotation type [GJSB05]. E.g., in the following EJB related example the declaration of the class Category is annotated with the annotation @Entity, whose member access is set to AccessType.FIELD; i.e., the container should access the entity's state using direct field access:

```
1 @Entity(access = AccessType.FIELD)
2 public class Category {...}
```

Listing 6.2: An annotated class

The use of such annotations often imposes certain implementation restrictions on the decorated program constructs. Consider, e.g., the java.lang.Override annotation of Java 5, which can be used to annotate non-abstract methods to state that they must override a method in a superclass. Since

[4]It is assumed that the reader is familiar with XQuery (cf. Section 5.1) and the XML representation of Java bytecode as generated by BAT$_2$XML (cf. Appendix V).

the Override annotation is defined as part of the language, the implied implementation restriction is enforced by Java compilers.

This is, however, not true for user-defined annotations. An example are the annotations defined by the EJB 3.0 specification. In EJB 3.0, components are Java classes annotated with the specified EJB annotations. Based on these annotations, the container will generate corresponding home and remote interfaces and extract the configuration information it needs.

However, the effect of annotating a bean with, e.g., Entity should go beyond driving the generation of its interfaces and providing configuration information to the container. It should also mean that implementation restrictions implied by the annotation, as explicitly stated in the specification, should be checked for. An example of such a restriction on an entity bean is: *"An enterprise bean must not use thread synchronization primitives..."*.

As in the EJB example, annotated program elements often have to follow constraints beyond those defined by the language's semantics. But, unfortunately these constraints are currently not checked at all or only to a very limited extent. Hence, a violation can remain undetected and result in deployment-time or even subtle run-time errors. Means to detect such violations using XQuery queries are discussed in the following.

6.3.1 Defining Implementation Restrictions

To illustrate the use of XQuery for the detection of violations of implementation restrictions, three analyses are discussed in the following.

Basically, each query just selects those elements which violate a restriction. Let us consider a simple check first. In section 6.1 of the EJB 3.0 specification [EJB05] [Requirements on the Entity Bean Class] it is stated that:

> *The entity bean class must not be final. No methods of the entity bean class may be final.*

A possible query to detect corresponding violations is shown in Listing 6.3. The first line selects all classes that have the javax.ejb.Entity annotation and stores the result in the variable $ebs. The variable $xirc:project-files is the set of all classes that are not defined in a library (in a ".jar" file). After that, the second line determines for all entity beans ($ebs) the set of classes and methods that are declared final.

```
1 let $ebs := $xirc:project-files/class[./annotations//@type ="javax.ejb.Entity"]
2 return $ebs[@final = "true"] union $ebs/method[@final ="true"]
```

Listing 6.3: Checking for Entity beans that are declared final

A second example concerns the dependency between annotations. Certain annotations can only be used in combination [CM04]. E.g., annotating a method with javax.jws.WebMethod requires that the class is annotated with javax.jws.WebService [Web05]. To check this dependency the query shown in Listing 6.4 first selects all classes that declare a method with the WebMethod annotation (Line 1) and then subtracts (Line 2) all classes that are annotated with the WebService annotation (Line 3). The set of classes that have WebMethods, but do not declare to be a WebService is returned.

```
1 $xirc:project−files/class[.//annotations//@type ="javax.jws.WebMethod"]
2 except
3 $xirc:project−files/class[./annotations//@type ="javax.jws.WebService"]
```

Listing 6.4: Checking dependencies between annotations

The queries discussed so far only analyze a class's interface, i.e., the class declaration itself and the declared methods. The analysis shown in Listing 6.5 also analyzes method implementations. The EJB 2.1 specification (which is referenced by EJB 3.0) states:

> *An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.*

The query to detect violations of this restriction checks that:

- no method is synchronized (Line 3)
- the `synchronize` statement is not used (Line 4) — `synchronize` statement manifests in `monitorenter` and `monitorexist` instructions at Java bytecode level
- none of the `wait` or `notify` methods is called (Line 5 – 8)

```
1 let $c := $xirc:enterprise−beans()
2 return
3   $c/method[@synchronized="true"]
4   union $c/method/code//monitorenter
5   union $c/method/code//invoke [
6           @declaringClassName="java.lang.Object"
```

```
7           and ( @methodName="wait" or
8                 @methodName="notify" or @methodName="notifyAll")
9   ]
```

Listing 6.5: Checking that no thread synchronization primitives are used

The queries discussed so far are self-containing, i.e., given the database the queries can directly be executed. However, many queries have identical parts when structural properties of classes are checked. For example, the queries to check an Entity bean's implementation restriction nearly always start with a path expression to determine all classes that are entity beans:

```
let $ebs := $xirc:project-files/class[./annotations//@type ="javax.ejb.Entity"]
```

These common parts can require a significant amount of a query's evaluation time: In case of a simple query up to 80–90% (as we will see in the evaluation). Hence, by factoring out the common part and executing it only once significant performance gains can be achieved.

Given MAGELLAN's extensible analysis stack and its open data model, and given the query chaining feature of the embedded XQuery engine (cf. Section 5.1.2 for details), the evaluation of queries in several steps is directly supported. For example, it is possible to execute a query that selects all EJBs (Listing 6.6) in a first step and to store the result in a fact, e.g., AllEJBs. Such queries, i.e., queries that determine the context for the evaluation of subsequent queries are called *context defining queries* in the following.

context defining query

```
1 /db:all/db:document[@type = "source"]
2   /class[
3       ./annotations//@type = "javax.ejb.Stateless"
4        or ./annotations//@type = "javax.ejb.Stateful"
5        or ./annotations//@type = "javax.ejb.Entity"
6        or ./annotations//@type = "javax.ejb.MessageDriven"
7   ]
```

Listing 6.6: Context defining query (all EJBs)

A query that checks that a specific property holds, e.g., that no finalize method is implemented by an EJB (Listing 6.7), then declares a dependency on the AllEJBs fact and uses the information as the context for its own execution. The query refers to the (external) context using XQuery's "." notation. This type of query is called a *c*ontext dependent query in the following.

context dependent query

```
1 ./method[@name="finalize"' and empty(./signature/parameter)]
```

Listing 6.7: Context dependent query (EJBs must not implement finalize)

6.3.2 Magellan Integration

As described in Section 5.1, each query is wrapped into a small Java wrapper that passes the XQuery to the query engine and post-processes the result(s).

Context-defining queries are directly executed against the database and the result — sets of references to elements in the database — are stored in a fact. A context dependent query is executed using the result of a previous context defining query as its evaluation context. The result of context dependent queries, e.g., violations of implementation restrictions, is then reported to the user.

For example, the Java wrapper of the query that selects all EJBs is shown in Listing 6.8. The wrapper extends the class ContextDefiningQuery (Line 2) and specifies the ASL file (Line 1) which defines the analysis' dependencies (cf. Listing 6.9). The class ContextDefiningQuery provides the MAGELLAN integration and manages the interaction with the embedded XQuery engine. In particular, MAGELLAN's interface for analyses that should be executed along with the incremental build process is implemented. Further, the result of a query is automatically stored in the specified facts. In case of Listing 6.8, the query defined in the file SelectEJBs.xq (Line 4) will be executed and the result will be stored in the fact: AllEJBsFact (Line 4).

```
1 @ASL{"SelectEJBs.asl"}
2 public SelectEJBsQuery extends ContextDefiningQuery{
3   public SelectEJBsChecker(){
4     super(AllEJBsFact.getId(),"SelectEJBs.xq");
5 } }
```

Listing 6.8: Java wrapper for the "select all EJBs" query

The ASL file of the "select all EJBs" query is shown in Listing 6.9. It specifies that the XML representation of the Java bytecode (CF_XML, Line 2) has to be available and that the analysis maintains the LSV entity AllEJBs (Line 3).

```
1 analysis SelectAllEJBs
2   reads XMLDB/CF_XML
```

```
3   maintains XMLDB/CF_XML/AllEJBs
```

Listing 6.9: ASL file of the "select all EJBs" qzuery

In Listing 6.10, an example of a checker that is implemented as a context dependent query is shown. The wrapper is identical to the previously discussed wrapper except that the class ContextDependentChecker is extended. As in the previous case, the class takes care of the MAGELLAN integration and the interaction with the query engine. However, instead of storing the results in a fact, error messages are generated for each result. Furthermore, the information stored in a fact by a previous analysis (Line 4) is set as the context for the query execution.

```
1 @ASL{"NoFinalizeMethods.asl"}
2 public NoFinalizeMethodsChecker extends ContextDependentChecker{
3   public NoFinalizeMethodsChecker(){
4     super(AllEJBsFact.getId(),"NoFinalizeMethods.xq");
5 } }
```

Listing 6.10: Java wrapper for the "no finalize methods" query

The ASL file of the "no finalize methods" checker is straight forward: It just specifies that the XML based representation of the Java bytecode is read as well as the fact which keeps references to all EJBs (Line 2).

```
1 analysis NoFinalizeMethods
2   reads XMLDB/CF_XML/AllEJBs
```

Listing 6.11: ASL file for the "no finalize methods" query

Due to the specified dependencies in the ASL files, MAGELLAN will first schedule the context-defining query and then the context-dependent query.

6.3.3 Evaluation

The use of XQuery for implementing checks of structural properties is assessed based on queries that check the constraints defined in the EJB 3.0 specification [EJB05]. The structural properties that are checked by the queries are described in Table 6.2. The queries were evaluated against a demo release of the xPetstore project [BKT+04] that was ported to EJB 3.0. The project consists of 46 classes.

The measurements were taken on an Intel Celeron 2.40 GHz system with 504 MB RAM running Windows XP, J2SE 5.0, Saxon 8.1 and Eclipse 3.1M2 as the underlying platform. Since the embedded XQuery engine (Saxon) is not thread safe the queries were executed sequentially and no performance evaluation on a dual processor system / dual core system was made.

The XML database had 2833 class entries, which consists of the classes belonging to the xPetstore project and all public classes and interfaces of all Java APIs delivered with Java 5. Classes in the javax.swing.*, java.awt.*, and in the com.* packages were exempt, as no classes defined in these packages were used. Additionally, all necessary JARs to compile the xPetstore project were included.

The evaluation of the original xPetstore project, which run without any error being signaled, required 1.97 seconds. To make the evaluation more realistic, we injected some problems into the project code. The evaluation of all 48 queries against the messed project code generated correctly 53 messages and was executed in 3.56 seconds. In both cases, the time required by Eclipse to recompile the source file and to update the Magellan database should be added, which amounts to another 1-2 seconds. To keep the XML data in memory approximately 40 MB are required.

Detailed execution times for each query are shown in Table 6.2. The table lists the times required by the queries that check properties related to:

- all types of EJBs (CommonEJB)
- session beans (SessionEJB)
- entity beans (EntityEJB)
- message driven beans (MessageEJB)

Furthermore, for each of these categories the total evaluation time and the time of the context defining query is depicted.

Short Description	Seconds
CommonEJB	$\sum$ 0.643225
context defining query	*0.023961*
an EJB must not start threads	0.017519
the signature of the call back method is invalid	0.069257
EJBs must be public and must not be final or abstract	0.004002
... continued ...	

Short Description	Seconds
the chosen transaction attribute cannot be used	0.011743
an EJB must have a no-arg constructor	0.010397
a business method must not start with "ejb"	0.012741
an instance that starts a transaction must complete the transaction before it starts a new transaction	0.385770
an EJB with bean-managed transaction demarcation must not use (get/set)RollbackOnly	0.007356
(get/set)RollbackOnly should be called only in bean methods that execute in the context of a transaction	0.044467
UserTransaction is unavailable to EJBs with container-managed transaction demarcation	0.011552
a TransactionAttribute can only be specified with container-managed transaction demarcation	0.012814
the finalize() method must not be defined	0.004736
EJBs should not handle concurrent access on their own	0.013553
a transient field must not have the specified type	0.004410
SessionEJB	$\sum$ 0.831755
context defining query	*0.204696*
business methods must be declared as public and must not be final or static	0.000767
the name of Session beans should have the suffix "EJB", "Impl" or "Bean", if the business interface should automatically be derived	0.000576
for update / delete operations a transaction context is required	0.047968
argument and return types must be legal types for RMI/IIOP	0.476183
argument and return types must be legal types for JAX-RPC	0.027315
Timers cannot be created for stateful session beans	0.000693
multiple business interfaces should be annotated as `Local` or `Remote`	0.046658

... *continued* ...

Short Description	Seconds
for a stateless session beans web service endpoint interface, only the Required, RequiresNew, Supports, Never and NotSupported attributes may be used	0.001131
this SessionContext's method cannot be called	0.024672
EntityEJB	$\sum$ 1.928637
context defining query	*0.147486*
the persistent field's type is invalid	0.463888
the `JoinColumn` annotation is needed, if the primary key values of the source and target entity are different	0.001352
every entity must have a primary key and the primary key must correspond to only one field or property of the entity bean class	0.006029
persistent properties with `@Basic` may not be an entity association	0.016634
a one-to-many association must be bidirectional; the target entity must have a matching many-to-one association	0.001210
invalid dependent class	0.159400
every entity bean must have a primary key	0.003133
a protected field is to be accessed by the defining class only	0.032637
an entity bean that is a subclass of another entity bean must have the same primary key	0.155760
entity beans must have getter/setter-methods for persistent fieds	0.099020
the methods of the entity bean class must not be final	0.003987
collection-valued persistent properties must have type java.util.Collection or java.util.Set	0.737543
invalid type for primary key	0.080830
MessageDrivenEJB	$\sum$ 0.015559
context defining query	*0.011637*
... continued ...	

Short Description	Seconds
for a message-driven bean's message listener interface, only the Required and NotSupported transaction attributes may be used	0.003602
message driven beans must implement the message listener interface of the messaging type	0.000319

Table 6.2: Evaluation times of queries

As the performance figures show, splitting the checking of structural properties in two steps leads to a significant performance improvement; without the two step process the time required by the context defining query would have to be added to each query. This would double the overall analysis time ($\approx 7secs.$). Nevertheless, the result of this evaluation shows that the analysis time is significant considering the project's size — approx. 3.5 seconds when checking all implementation restrictions. Hence, running XQuery based checkers regularly along with the incremental build process is only feasible for very small projects. However, the analysis time is reasonable fast to execute the XQuery based analyses on-demand. In this case analysis times of multiple seconds or even a few minutes are acceptable.

6.4 Conclusions

Based on the evaluation of Magellan for developing static analyses, the following conclusions can be drawn:

- The developed analyses and checkers demonstrates that Magellan enables the definition of a wide range of different analyses. Analyses are supported that — at least — range from simple analyses of structural properties up to sophisticated intra-procedural data- and control-flow analyses. Based on these promising results it is expected that Magellan also supports inter-procedural analyses. However, whether it is possible to implement sophisticated inter-procedural analyses such that they can be executed along with the incremental build process remains an open issue. Exploring this issue is left for future work, because it is mainly related to the performance of the analyses and it is not related to the concept of open static analysis platforms as such.

- The performance evaluation shows that MAGELLAN enables running a larger number of analyses along with the incremental build process. Executing all 30 BAT based checkers does not cause an overhead that can be perceived by a developer when editing its source code ($\approx$ 100 milliseconds). Only if multiple resources (classes) are affected, e.g., in case of refactorings, the analysis time may be perceivable. Since such changes are not performed frequently, a small overhead of only a few seconds is acceptable.

- The automatic parallelization of the analyses as done by MAGELLAN is effective. Without fine-tuning of MAGELLAN's implementation a 28% performance improvement on a dual core system compared to a single core system is achieved. Furthermore, the parallelization is fully transparent for developers of analyses and checkers. Hence, the development of checkers is simplified since a manual parallelization of analyses is not necessary.

- XQuery has proven to be well suited for the development of checks of structural properties; a large number of different checkers were easily implemented. However, for non-trivial programs the performance of the XQuery based checkers is not sufficient for running them as part of the incremental build process.

  The insufficient query performance is due to the lack of any support for incremental query evaluation. To determine the result of a query the query engine always evaluates queries w.r.t. the whole database. The query engine does not reuse a query's result and a description of the latest changes to update the result of the query. However, the XQuery based checkers are fast enough to be executed on-demand or, e.g., automatically when the user wants to check in source code into a version control system.

- Domain specific annotations, e.g., the EJB annotations, are well suited as hooks for checkers of structural properties. All implementation restrictions defined in the EJB specification are directly related to annotated elements. Hence, if a standard for open static analysis platforms would exist, libraries and frameworks can be envisioned that are delivered with checkers for the libraries' implementation restrictions. If the user then makes use of the library and uses the library's annotations the corresponding checks would automatically be executed and pinpoint the developers to violations.

- The approach underlying MAGELLAN has proven to be very flexible. E.g., supporting the evaluation of XQuery based queries in two steps did not require any changes in MAGELLAN or in the embedded XQuery engine. The evaluation process is implemented leveraging MAGELLAN's open data model and the support for the definition of dependencies between analyses.

To sum up, MAGELLAN has proven to be very flexible and to enable the definition of lightweight static analyses using different techniques. Furthermore, it was shown that the performance that can be achieved using MAGELLAN is sufficient for day-to-day work when several checkers are executed regularly.

Chapter 7

Software Comprehension

This chapter shares some material with: *Comprehensive Software Understanding with* SEXTANT [EHMS05].
A detailed study of the software exploration capabilities of SEXTANT is published in *The Sextant Software Exploration Tool* [SEHM06].

7.1 Introduction

To maintain and extend software systems developers need to understand a software's internal structure to ensure that changes do not break the intended behavior of the system [vGB02]. Unfortunately, appropriate documentation of the system's structure is often missing or outdated; even when accurate documentation is available, tool support for software comprehension is indispensable, given the complexity of today's software systems.

In general, tools for software comprehension can be classified in two groups. First, *software visualization tools* providing visualization techniques for a software system [DDL99, KC98, MTW93, SM95, LD01, SWFM97], e.g., CodeCrawler [DDL99], or SHriMP [SM95]. Second, *software exploration tools* that provide means to navigate along a software system [CFKW95, Fav01, JD03, RSK00, SCHC99, SLVA97], such as tksee [SLVA97] or JQuery [JD03]. In the following, the focus is on software exploration tools since studies indicate that the navigational aspects are very important in software comprehension tools [BK01].

To comprehend a given application developers search for elements in a software system over and over again and navigate through their relations. During this exploration process, also called *Just in Time Comprehension*

[SLVA97], the developer constructs a mental map of the visualized information. This process is supported by comprehension tools by providing explicit means to explore the relations between different source elements. For example, given a class the developer can directly navigate to the declared methods and continue its exploration by navigating to the callers, callees or accessed fields.

To support code comprehension and to make an initial assessment of MAGELLAN as a platform for building software comprehension tools, the SEXTANT code exploration and navigation tool was developed. In Section 7.2, general requirements on software exploration tools are discussed. After that, in Section 7.3, the SEXTANT tool is presented and evaluated w.r.t. the identified requirements. This chapter concludes with an assessment of MAGELLAN as a generic platform for building software comprehension tools. The assessment is based on the experiences gained while implementing SEXTANT's functionality related to the identified requirements.

7.2 Requirements on Tools for Software Exploration

The following requirements were derived based on the analysis of well known software comprehension and exploration tools as well as related literature, e.g. [DDL99, KC98, MTW93, SM95, LD01, SWFM97, CFKW95, Fav01, JD03, RSK00, SCHC99, SLVA97, BK01, SWM00, Bro83, vMV95, LPLS87]. The requirements are summarized in Table 7.1 at the end of this section.

SET-R1 Integrated comprehension

Three basic software comprehension strategies have been described in [SWM00]: Bottom-up, top-down, and mixed strategies. Developers using a bottom-up approach achieve a high-level software comprehension by starting to read the low-level source code and stepwise abstracting from it. Software engineers following the top-down approach [Bro83] use their general domain knowledge to formulate an initial hypothesis about the software. The initial model is then refined by trying to verify it and by searching for corresponding structures in the code. Mixed strategies assume that developers are capable of using both aforementioned strategies [LPLS87, SLL+88]. Von Mayrhauser et al. [vMV95] present the integrated model of software comprehension, a

refined mixed strategy, in which developers use bottom-up and top-down approaches at different abstraction levels, frequently switching between them. Following on this work, it is required that software comprehension tools support an integrated comprehension subsuming both, the bottom-up and top-down approach.

SET-R2 Cross-artifact support

Modern tools, such as persistence frameworks and component technologies, e.g., Enterprise Java Beans (EJB) [EJB03], aim at better mastering the software complexity, ironically also causing a new kind of complexity to emerge. The developer using such technologies is forced to work with a multitude of different kinds of artifacts: Besides source code, a large number of external libraries is often used and information that affects the runtime behavior is stored in XML or properties files. As a consequence, it is no longer possible to analyze and comprehend a software project without considering and aggregating information contained in all kinds of artifacts.

Recently, the industry becomes aware of the additional complexity caused by using different artefacts for specifying an application's runtime behavior and tries to remedy the situation by using meta-data specified along with the source code. The meta-data is meant to replace the usage of XML and properties file. For example, the Enterprise Java Beans specification 3.0 [EJB05] advocates the usage of Java 5 annotations to specify — amongst others — the type of a bean, the transaction attributes and security attributes. However, a large number of applications that make use of very different artifacts already exists and need to be extended and maintained for many years to come. Further, many other frameworks will continue using XML and related files for configuring runtime environments. In the context of service-oriented architectures, e.g. when developing WebServices, XML is inevitable anyway. Hence, exploration tools need to enable software comprehension across artifact borders.

SET-R3 Explicit representation and referential integrity

To support the creation of a mental map of the software all of its elements and the relations between those should be visualized explicitly and the referential integrity among them should be maintained as the navigation process unfolds. Unfortunately, this is not the case in many mainstream

IDEs. For instance, in Eclipse [Ecl06] it is possible to use hyperlinks to get from one software element to another: One can navigate from a class to all its methods and from each method to all of its callees, and so on. The path followed by such an exploration is, however, not visible, making it hard to build the mental map [JD03]. By using specialized views such as the call hierarchy view, it is possible to see the path for a single kind of relationship. But, switching to a different kind of relationship requires switching the view and thus can cause disorientation [SFM97].

Furthermore, the navigation often lacks referential integrity in the sense that the same element may appear several times potentially in different views of the IDE. For instance, assume a method m1 is called by two other methods m2 and m3. In Eclipse, one can use hyperlinks to navigate from m2 and m3 to m1. However, in this process m1 will appear twice in two apparently unrelated views – the list of methods called by m2, respectively by m3. The referential integrity of m1 is not maintained during the exploration and it is hard to discover that m2 and m3 are related by the property of calling m1.

SET-R4 Extensibility

Software exploration often involves navigating to common software elements and along common relations. For instance, common elements one would like to navigate to while exploring a Java application include classes, methods, and fields; common relations frequently used to navigate along are inheritance and call relationships. In addition to such common elements and relations, specific application domains or specific libraries in use may require to navigate to new kinds of elements and along new kinds of relationships. In, for example, an EJB project technology specific relations between a class, its corresponding public interfaces and related elements in deployment descriptors become relevant relations to navigate along. However, tool developers cannot foresee all contexts in which an exploration tool is used. Hence, software exploration tools should be extensible to accommodate for domain-specific navigation elements and relationships as needed.

SET-R5 Traceability

In most cases, comprehension is not an independent software development task. Rather, we use exploration to understand a system to continue with a modification, which is done at the code level. Hence, the ability to switch

instantly and seamlessly between the graphical notation and the corresponding source code is essential for practical use [BK01]. Furthermore, often the exploration process might only give us hints; the actual understanding or the validation of our hypotheses may require switching to the source code representation of the system.

Requirement		Description
SET-R1	Integrated comprehension	Integrated and simultaneous support for bottom-up and top-down software comprehension strategies.
SET-R2	Cross-artifact support	Navigation across different types of artifacts.
SET-R3	Explicit representation and referential integrity	Explicit visualization of the navigation path and referential integrity between explored entities.
SET-R4	Extensibility	Support for user-defined queries.
SET-R5	Traceability	Navigate between the graphical model and the code.

Table 7.1: Requirements on software exploration tools

7.3 Code Exploration and Navigation with Sextant

Sextant is a cross-artifact software exploration tool that enables developers to browse a project's sources using predefined queries as well as user defined queries. The results of all queries are integrated into one graph that represents all explored elements and the relations between them. The graph is automatically layed out and shown to the user.

SEXTANT extensively uses MAGELLAN's XQuery interface for searching program elements and browsing along different kinds of relations. These two navigational styles support bottom-up and top-down comprehension. The integrated comprehension model is enabled by an integrated view: The elements discovered in any search or browsing activity are visualized as nodes in a graph, while relationships between elements are depicted as edges of the graph, whereby, all elements and relationships are explicitly represented. By

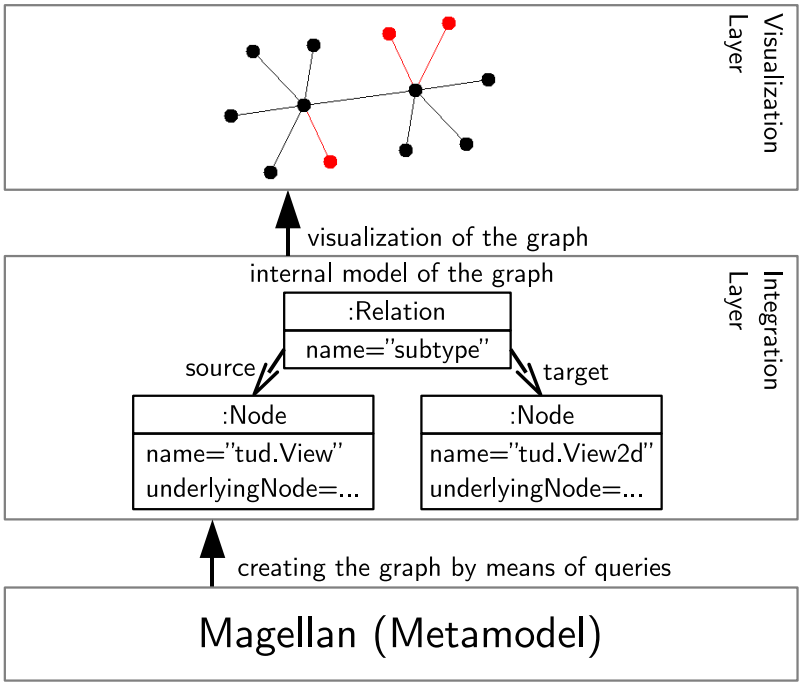

Figure 7.1: Conceptual Model of Sextant

building upon Magellan, Sextant offers functionality to switch between the graphical notation and the source code representation. Furthermore, Sextant is extensible with new node types, and user-defined relations.

7.3.1 Architecture

Sextant has the three-tier architecture shown in Figure 7.1, comparable to the proposal described by Lanza [Lan03]. The *metamodel* provides the raw data and means to extract information. The metamodel is realized by Magellan and the XQuery interface is used to extract the information. The *integration layer* uses the information of the executed queries to construct a graph which represents the relations between the program elements explored by the user. The graph is visualized by the *visualization layer*.

7.3.1.1 Metamodel

As its datamodel, Sextant uses the XML database of embedded XQuery engine, i.e., it uses BAT$_2$XML's (cf. AppendixV) representation of Java class files as well as the result of an analysis that adds all ".xml" documents of a

project to the XML database. To extract information, SEXTANT uses both types of queries supported by MAGELLAN's XQuery interface (cf. Section 5.1):

- Queries to search the entire database are used, for example, to find type definitions, methods, or declarations of Enterprise JavaBeans (EJBs). These queries provide the search capabilities.

- Context dependent queries; i.e., queries which are defined with respect to an XML node in the database, are used to enable the project's exploration. For example, the context of a query to get all fields accessed by a method is a reference to a method node in the XML database. The results are only those fields accessed by the particular method.

The result of both types of queries — search queries or context-dependent queries – can be either references to other nodes of the XML database or derived information, i.e., any information that is not directly stored in the database. For example, a query that calculates the depth-of-inheritance metric returns derived information.

7.3.1.2 Integration Layer

The integration layer builds a graph from the results of the executed queries.

A node of the graph is the representation of a software element or a derived information returned by previously executed queries. Each node that does not represent derived information has a reference to the element it represents; this reference is used to ensure that each element of the model is represented by at most one node in the graph, even if the same element is selected by multiple queries.

In general, the result of search or context-dependent queries are either XML elements or XML attributes. But, to enable context dependent queries that are only defined w.r.t. specific nodes or to enable different visualizations of the graph's nodes later on, the XML elements are categorized. Example categories are: MethodDeclaration, FieldDeclaration, or EJBDeployment-Descriptor. For categorizing elements a query is executed where the node's underlying XML element is set as the context. The result of the query is the element's category, also referred to as the node's type in the following. Basically, a node's type is just a simple unique name used to identify nodes which represent similar elements. The query which types the elements can be extended to handle new kinds of XML elements.

The node's type is in particular used to support context dependent queries which are only defined with respect to a well-defined set of node types. In this case, a node can be used as the context of a query, if and only if the query explicitly supports the type of the node. While each node has exactly one type, a query can be defined with respect to multiple node types, e.g., a query to get the declaring class is well-defined for method declarations and field declarations.

An edge of the graph represents a relation between two nodes exposed by executing a query; hence, the semantics of an edge is solely determined by the query. An edge always points from the node that was set as the context for the query evaluation (the source node) to the nodes that were returned by the evaluation (the target nodes).

7.3.1.3 Visualization Layer

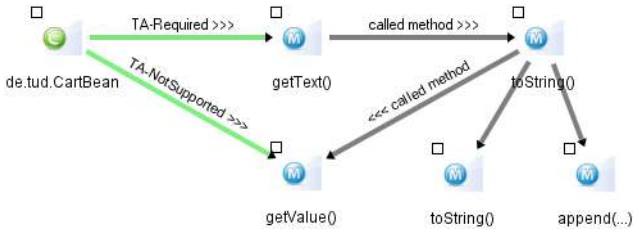

Figure 7.2: Visualization of cross-artifact based relations

The visualization layer visualizes the graph constructed by the integration layer and enables navigation through the software project. It is based on TwoMore [DEO+05], a tool originally designed for the manipulation and visualization of topic maps. Software is represented by a graph with source elements as nodes and relations as directed edges between the nodes. For instance, in Figure 7.2 the class de.tud.CartBean and the method getText() are elements of the software and they are related by means of the TA-Required relationship. Given a net of nodes shown on the screen, selecting a node will show up a menu with the list of all applicable queries. The user can then choose the query for the further exploration. The result of evaluating a selected query will be integrated into the existing net.

The visualization layer can be adjusted in various ways. For instance, it is possible to choose between different layout algorithms. E.g., a hierarchical layout orders elements in a tree-like structure, the spring force layout instead

automatically places related elements in a concentric circle around the source node. To keep parts of the graph structure fixed, one can explicitly assign a fixed place to one or more nodes. Fixed nodes have the advantage that they will not be rearranged when additional nodes are added to the net and therefore ease the comprehension.

Information about the visual appearance of an edge is specified in a query's meta-data. For instance, it is possible to set the color and the description of an edge.

The visual appearance of a node is determined by the node's type and it is possible to specify the color and icon that is to be used. Available visualization options are shown in Figure 7.3.

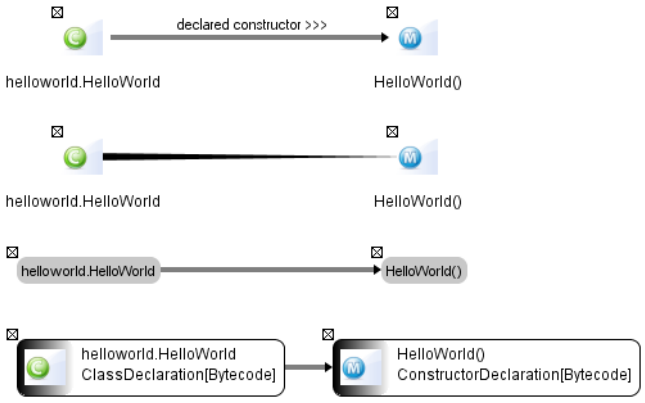

Figure 7.3: Visualization options provided by SEXTANT

In the first case, the type of the node is indicated by an icon and the edge is explicitly labeled. In the second case, a spike is used instead of an arrow: If the relation between two nodes is obvious this representation is more concise. The third case is the most compact one and is particularly useful if the nodes have the same type and a large net is to be explored, as we will see in the evaluation section. The last case is the most elaborate version: The type, color and icon of the node are explicitly shown. Visualizing the type of a node proved helpful for comprehending the structure and the types of relations.

The node's description, e.g., *"helloworld.HelloWorld"*, is determined by executing another query that is dependent on the node's type and where the underlying element is set as the context. For instance, given the XML snippet shown in Listing 7.1 the query will return "**int** hashCode()".

```
1 <method name='hashCode'>
2   <signature>
```

```
3     <return type='int'/>
4   </signature>
5 </method>
```

Listing 7.1: XML representation of a Java method's signature

7.3.2 Evaluation

The following two case studies demonstrate how SEXTANT was used to understand the source of a bug, respectively to discover overly complex structures which led to refactorings.

In the first case study, a small EJB project is explored to demonstrate the need for the integrated comprehension strategy (*SET-R1*). Furthermore, this case study also illustrates the need for cross-artifact support (*SET-R2*) and extensibility (*SET-R4*). In the second case study, SEXTANT is applied to understand parts of a complex application, namely the Steamloom Java virtual machine [BHMO04]. The goal is to analyze the dynamic weaving control flow and to derive refactorings from the visual representation. This second case study will demonstrate how the explicit representation of all elements and relations (*SET-R3*) as well as links between the graphical representation and the corresponding source elements (*SET-R5*) are crucial for our needs.

7.3.2.1 Exploring an EJB Project

As an introductory example, the usage of SEXTANT to explore a small EJB project is presented. Consider a scenario in which a developer receives a bug report for an EJB component with the ejb-name CartBean, indicating problems with the transaction handling. Using this component resulted in the exception: TransactionNotSupportedException. Given this information, the developer assumes that the problem is related to the CartBean and starts searching for the class that implements the bean with the specified name using a corresponding query. The result is the class de.tud.CartBean, represented by the left-most node in Figure 7.2.

Next, the developer executes a query on this node to get all methods defined by the class that have the transaction attribute NotSupported. This query returns the node representing the getValue method. A further query to get all methods called by getValue does not return any further results.

After going back to the node of the CartBean class and executing another query to get all methods with the transaction attribute Required, the getText

method is returned. Again, the developer wants to further explore the call graph and executes the called methods query on this node, which returns the toString method. After executing the same query once again for the toString method, the developer discovers that the getValue method gets called from within toString, hence, finding a circle in the call graph. At this point, the developer spots a severe problem: A method that always runs in a transaction context calls a method that does not support transactions.

Based on the given example, we now discuss the importance of the proposed requirements for software exploration tools.

SET-R1 This example illustrates the usefulness of a seamless integration of top-down and bottom-up comprehension. The basic top-down and bottom-up comprehension approaches are supported by means of the searching, respectively browsing facilities of SEXTANT.

Combining the two navigational approaches and switching between them is facilitated in that a single graph-based visualization is used to represent the results of a search as well as the elements a developer browses to. In the example, first a search query to find the implementation classes of beans with a certain ejb-name was executed; subsequently, the exploration continued by browsing the result of the search.

Furthermore, different views resulting from different queries and representing the system at different levels of abstraction, are visualized in a single graphical representation. For instance, the search query revealed the implementation of an EJB component while another query built the call graph for a method. Because those different views provide complementary information, developers need to be able to navigate among them to completely understand an architecture [KC98]. With SEXTANT, it is possible to *fuse* different views by using different kinds of relationships in each step of the navigation.

SET-R2 The EJB nature of the sample project illustrates the need for navigating across different kinds of artifacts. In the example, the queries used information from Java class files as well as from XML based deployment descriptors. Without an explicit support by software exploration tools, those kinds of relations have to be established manually. First, this can be very time-consuming. Second, it runs contrary to the comprehension process, because not all elements and relations are visualized and one can easily get lost during the exploration [JD03].

SET-R4 Last but not least, this case study also demonstrates the need for extensibility. To explore the discussed EJB project, domain specific queries were required. Given SEXTANT's generic Java related queries it was not possible to successfully explore the Project; cross-artifact reasoning is required and the semantics of an EJB deployment descriptor's elements needs to be understood. Hence, extensibility is crucial for defining project and domain specific queries.

For illustration, consider how this feature was used in the case study: For exploring the EJB application, a new query was written. The query is shown below and determines the implementation class for a bean (with its name stored in the variable $ejb-name). In natural language this query reads as follows: First, save the name of the implementation class in the variable $class-name. Then return the class with this name.

```
1 let $class−name := //ejb−jar/enterprise−beans/
2      (session|entity)[./ejb−name = $ejb−name]/ejb−class/text()
3 return //bat:class[@name = $class−name]
```

Listing 7.2: XQuery to get the Java class given a bean's name

7.3.2.2 Analyzing a Java VM

Steamloom [BHMO04] is a Java virtual machine with native support for aspect-oriented programming [KLM+97] implemented as an extension of the Jikes Research Virtual Machine (Jikes RVM for short). Steamloom consists of roughly 200,000 lines of Java code, about 150,000 of which belong to the underlying Jikes RVM and other 30,000 belong to the bytecode toolkit used for implementing the weaving functionality.

One of the features that Steamloom adds to the Jikes RVM is dynamic aspect weaving capabilities. The control flow inside the weaving component is complex; to understand it, several queries were applied to the control flow that is initiated by the VM_AspectUnitRegistry.weaveIn() method. This method is responsible for weaving aspect functionality into one particular point in the application code. It calls one of three entry points of the CodeGenerator class: (a) generateCode(...), (b) generateAfterExecutionCode(...), or (c) generateAfterCallCode(...). The entire subsequent code generation control flow takes place inside the CodeGenerator class, by invoking several private methods. The calls to methods *outside* the CodeGenerator have been filtered

out using a dedicated query. This facilitates to keep focused on the intra-class control flow.

By evaluating a query to find the called methods several times, the entire weaving control flow within CodeGenerator was visualized (see Figure 7.4). By observing certain patterns in the visual representation of the control flow, it was possible to spot places in the weaving logic under exploration that had *bad smells* [Fow99]. Some of these observations led to refactorings. After performing the refactorings the evaluation was repeated using the same queries to rebuild the modified control flow graph for weaving. The result is shown in Figure 7.5: The number of method calls has obviously decreased, and the overall control flow is more clear. In the following, the discovered bad smell is described and the refactorings that were conducted. The various places in which refactorings were applied are marked by indexes in the two figures. Indexes in Figure 7.5 mark the results of refactorings applied to the respective locations in Figure 7.4.

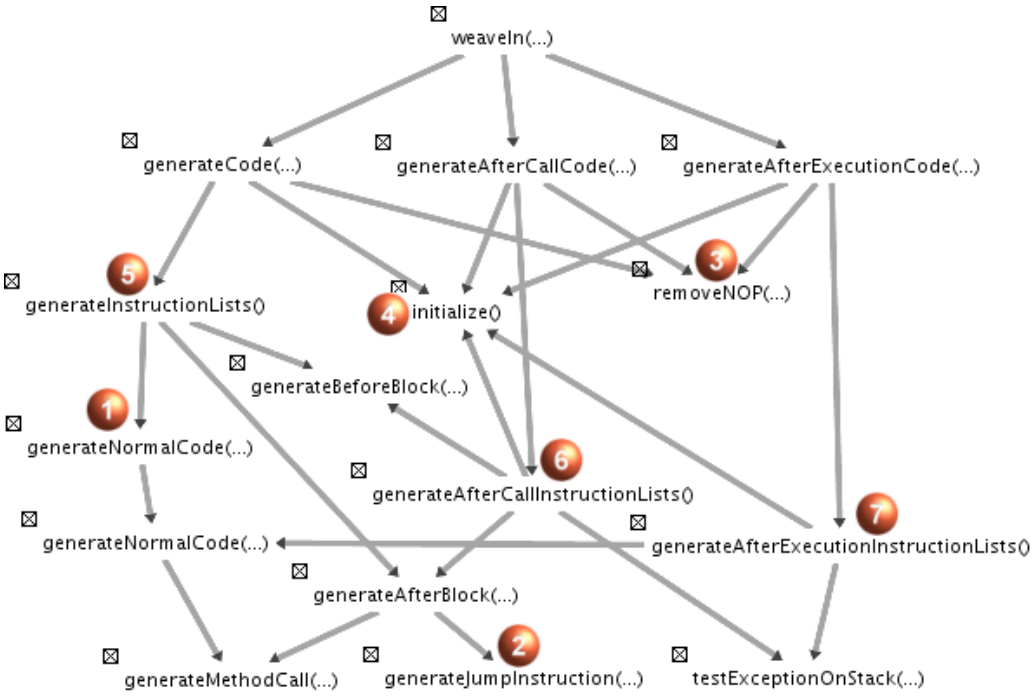

Figure 7.4: Weaving control flow of Steamloom before refactorings

One of the visual patterns can be seen at Index 1 in Figure 7.4. The node representing the method generateNormalCode(...) has only one incoming and outgoing arrow, i.e., it is called from one method only and calls only one

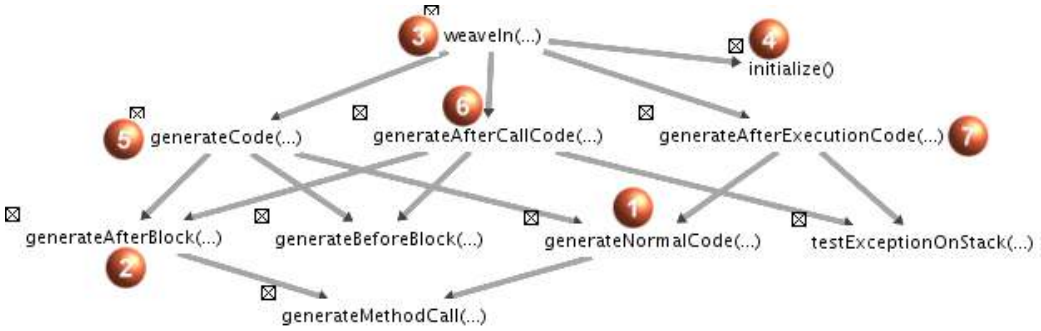

Figure 7.5: Weaving control flow of Steamloom after refactorings

method, namely another version of generateNormalCode(...) with one more parameter. Switching to the source code reveals that this method merely forwards a default value to the called method. Since these methods are private, the control flow can be simplified by deleting the forwarding version of generateNormalCode(...) and directly passing the default value of the second parameter from the actual call site.

The method generateJumpInstruction(...) (Index 2) also has only one incoming arrow, meaning that it is only called from one site. Given that the method is rather trivial, it was inlined at that call site.

Another method, removeNOP(...) (Index 3), is called from each of the three entry point methods. Since it has no outgoing calls to other CodeGenerator methods, one might suspect that this method is used as a mere service provider that does not further contribute to the actual control flow. Indeed, by taking a closer look at the source code, it was observed that the method was invoked exactly once from each of the entry points. This invocation took place just before a list of instruction objects was returned to the caller, namely the weaveIn() method. Since removeNOP(...) was rather trivial, this functionality was inlined in weaveIn(), right after the invocation of one of the three CodeGenerator entry points.

The initialize() method (Index 4) is called by several generate*() methods. Given that initialize() does not accept any parameters and some of the calling methods invoke each other as well, this looked like a good opportunity for clarifying control flow. Indeed, initialize() is a simple set-up method that assigns initial values to some state variables of the CodeGenerator. It was easily possible to move the call to initialize() to the weaveIn() method before the actual code generation control flow is entered. Each of the methods with indices 4, 5 and 6 is called from exactly one other method and could therefore

be inlined at the respective call site.

As in case of the previous case-study, the following paragraphs discuss the importance of the proposed requirements with respect to this study.

SET-R3 This case study illustrates how an integrated view for searching and browsing avoids the need to switch between different tools and views, which often causes disorientation. Integrated views can help to create and retain the mental map and navigate through the system without getting lost. Furthermore, the case study demonstrates how the graph-based visualization of SEXTANT directly represents the developer's exploration path with all elements and relations discovered in the exploration, whereby preserving referential integrity.

In fact, SEXTANT extends the notion of *not getting lost* as introduced by Janzen and de Volder [JD03]. By using a graph for the visualization instead of a tree structure as in [JD03], complex relationships between different information sources become apparent. For example, browsing to the callees from the nodes representing the entry point methods during the exploration of the weaving control flow in Figure 7.4 results in the method initialize() in all three cases. Using a tree-based visualization, the method would appear in three different subtrees and the developer has to establish the relationship manually by matching the names.

On the contrary, with the graph-based visualization of SEXTANT, the node for the initialize method appears only once in the exploration graph with incoming edges from all three entry points of the weaving process. Thus, graph-based visualizations can improve the comprehension of a software's inner structure by making relationships among single elements explicitly visible. This explicit representation was crucial for understanding the control flow and ultimately led to refactorings and simplifications in the code.

SET-R5 Further, this case study motivates the requirement *SET-R5* from the introduction, concerning traceability between the graphical representation and the corresponding source code. The graphical representation was well-suited to detect visual patterns indicating possible structural code smells. However, to judge whether a refactoring is appropriate, one needs to look into the source code.

For instance, when looking at the graph it can be seen that the method generateNormalCode calls a homonymous method with one more param-

eter, but to pinpoint the method as a forwarder it is necessary to look into the code. Due to SEXTANTs code link feature, one can synchronize the graphical representation with the code editor. Just after selecting any program element, the editor shows up the corresponding location in the source code. This enables to switch quickly between the different representations and further improves the comprehension, because one can also see low-level implementation details.

7.3.3 Related Work

Many tools have been developed to support the understanding of software systems. One category of software comprehension tools focus on the visualization facilities [DDL99, KC98, MTW93, SM95, SWFM97]. Two well-known examples are Rigi [MTW93] and SHriMP [SM95]. Rigi is a system for reverse engineering, primarily capable of the identification of subsystems by certain criteria, e.g., file containment or element names. The results of the identification are then visualized. All subsystems form a hierarchy, which is displayed in an overview window, but the details of a subsystem, i.e., the contained elements, are represented in their own window. Thus, Rigi follows a multi-window approach. SHriMP, a tool based on Rigi, provides an alternative visualization. All subsystems are represented in a single view using nested graphs. Along those, developers can navigate down to the source code. Besides this, well-known visualization techniques such as fisheye-view or pan and zoom are also available.

In contrast to SEXTANT with its lightweight visualization, both tools provide more complex visualization techniques. But, recent studies revealed that developers are often swamped with too many elements [Lan03, SWFM97] and too complex visualizations [SWM97]. For example, the existence of multiple, non-integrated views can cause disorientation as in case of Rigi, whereas the SHriMP visualization can result in an information overload. The proposed approach differs in such that not the whole system is visualized, but the developer explores software elements of interest step by step. Though, Rigi and SHriMP provide support for source code navigation using hyperlinks and for context navigation, the browsing capabilities are limited due to a small number of queries and the absence of means to add new queries or to customize existing ones. SEXTANT is fully extensible and tightly integrated with Eclipse and enables developers to switch seamlessly to the source code.

Other tools aim at combining the two navigational styles searching and browsing — a prerequisite to support the integrated model of software com-

prehension. Examples of those tools are Hy+ [MS95], Ciao [CFKW95], The Searchable Bookshelf [SCHC99], and SPOOL [RSK00]. All these tools have in common that they are based on a kind of repository, e.g., a fact base or a database, and provide advanced query mechanisms allowing a developer to extend the tool by defining new queries. Although this enables to search or browse along diverse relationships, one cannot fuse different views. If a developer uses for instance SPOOL and starts its exploration with a query, it is common that the next step will be the further evaluation of the results. Even though it is possible to make the results of the former query starting points for a new query, one looses the exploration path which is essential to build up a mental map of the software system.

On the contrary, SEXTANT presents the results of a complete exploration in a single view. This results in an explicit representation of the exploration path, preventing developers from getting lost during the exploration.

Feat [RM02] is a tool to create and manipulate representations of concerns. Developers can browse along different semantic relationships between program elements and add elements of interest to a concern. All concern elements and their interrelations are abstracted in a concern graph representation. SEXTANT's capabilities to search, browse, and visualize program elements are more advanced compared to Feat with its fixed program model. SEXTANT provides means to search and browse along different semantic relations and in different kinds of artifacts. Furthermore, the graph-based visualization is more appropriate for understanding interrelations between program elements than a tree-based one. However, the main contribution of Feat is a mean to make concern descriptions explicit using concern graphs.

The tool most similar to SEXTANT is JQuery [JD03]. It combines the advantages of query-based tools and hierarchical browser tools. Queries provide means to search for elements in the system on the one hand, and to explore code in terms of different kinds of relationships on the other hand. The results are visualized in a tree-based, hierarchical representation. Each resulting element can be used as the source for a new query. The results of this query form a subtree of the source element. Thus, the whole tree is an explicit representation of the exploration path. However, due to the hierarchical nature of JQuery's visualization, a single element can occur several times in different subtrees of the exploration. Each occurrence symbolizes a relationship between the element itself and its parents. However, to see all relationships of the elements the developer has to derive this knowledge manually by searching for all its occurrences in the tree. Another shortcoming is that JQuery is only capable to explore the Java structures in a software

system. The tool is not intended to integrate other kinds of artifacts, which restricts the applicability in modern software projects.

In contrast to JQuery, SEXTANT uses a graph-based representation which improves the comprehension of the relationships between the systems elements. Each program element occurs at most once in the view. When a new element is discovered and there are relations to other elements in the view, they get automatically visualized so that each relation is made explicitly visible. Furthermore, MAGELLAN is used as the data layer. This enables to store and query different kinds of artifacts in a uniform way. Those cross-artifact query capabilities broaden the scope of possible applications and enable developers to write domain- or technology-specific queries even if not only Java source code is used.

A different approach of software exploration, namely back-packing exploration, is described by Favre [Fav02]. The work is similar to SEXTANT by providing means to explore different kinds of artifacts. While SEXTANT uses Magellan's XQuery interface for queries, the G^{SEE} back-packing framework provides a generic successor interface with a single method returning all related elements for a given one. This enables the integration of various information sources. For instance, one can use the interface to integrate an object-oriented database or one could create an implementation of it using Java introspection to find related elements for a Java class. The simplicity of this interface facilitates the usage of existing libraries as new kinds of information sources with almost no preparation efforts. Hence, the meta-model can be elaborated during the actual exploration by integrating new source components interactively, which promotes the discovery of new concepts on-the-fly.

7.4 Conclusions

From the evaluation follows that MAGELLAN facilitates the development of first-class software comprehension tools. In particular, building comprehension tools on top of MAGELLAN has the following advantages:

- MAGELLAN enables researchers and developers who build software comprehension tools to focus on the integration and visualization layers, i.e., to focus on the functionality which distinguishes the comprehension tool from other tools. Functionality to parse the code and to maintain a model of the software is provided by MAGELLAN or by the set of

analyses already delivered with MAGELLAN. Hence, w.r.t. the architecture proposed for software visualization tools in [Lan03], MAGELLAN can be regarded as a first-class implementation of the *metamodel*. The XML view on the project's artefacts and the integrated XQuery interface which operates on top of the view provide the metamodel's functionality.

- The integrated XQuery interface supports the types of queries required by software comprehension and exploration tools: First, queries to search the data base. Second, queries that are defined w.r.t. a specific context element and which use the result of a former query as the starting point. Taken together these types of queries support bottom-up and top-down comprehension strategies. Hence, except from a user interface, the core functionality necessary to support integrated comprehension (requirement *SET-R1*) is available.

- The feature of the XQuery engine that queries return direct references to nodes in the database makes it particularly easy for software comprehension tools to satisfy the *referential integrity requirement* (part of *SET-R3*). To identify previously explored elements a simple reference comparison is sufficient.

- Since the embedded XQuery engine can execute any valid query that is passed in as a String object, the extensibility requirement (*SET-R4*) is also well supported. Software comprehension tools built on top of MAGELLAN just need to provide a user interface to support user defined (ad-hoc) queries; the core functionality is already provided.

- Base functionality for handling different types of artifacts is already included in MAGELLAN and is leveraged by the embedded XQuery engine to explicitly support cross-artifact queries (requirement *SET-R2*). To extend the set of artifacts that can be queried, it is sufficient to add analyses that map the artifacts to corresponding XML representations. After that it is immediately possible to execute cross-artifact queries.

- Efficient querying capabilities are readily available. The performance of MAGELLAN and in particular of the provided XQuery interface is sufficient to explore — at least — reasonable sized projects ($\approx$ 200.000 lines of code in the second case study). In all cases the queries are evaluated in less than one second and, hence, are fast enough for interactive explorations. Further, the time required to maintain the database along

with the incremental build process is negligible and does not lead to an overhead perceived by the IDE's users. To keep the database in main memory $\approx$ 100MB were required, which is still reasonable — given the configuration of current developer systems.

- Since MAGELLAN and the tools build upon it are tightly integrated into Eclipse, the developers can directly use the gained knowledge to adapt, correct or extend the source code of the system. After performing the change the underlying database is automatically brought up-to-date and the user can immediately continue the exploration.

- By building comprehension tools upon MAGELLAN the end user's effort to create an initial configuration for the code comprehension tool will be minimal when compared to other tools. E.g., as reported in [DDL99], some reverse engineering tools require an initial configuration that requires up to two days before the tool can correctly parse the project's source code and eventually build up the model of the software. Due to MAGELLAN's Eclipse integration this overhead is avoided.

- The end-user can use multiple software comprehension tools simultaneously as the time required during incremental builds to maintain the source model is independent of the number of used software comprehension tools. In case that all tools have the same requirements the build time overhead is identical to using one tool only. In case of differing requirements MAGELLAN still reduces the overall overhead as those analyses that are executed to satisfy the overlapping requirements will be executed only once. Hence, it is at least more likely that multiple tools can be used.

Chapter 8

Assessing the Quality of Code

This chapter shares some material with: *QScope: an Open, Extensible Framework for Measuring Software Projects* [EGM+06]

8.1 Introduction

Assessing the quality of code is important to answer questions such as: "Is it necessary to refactor the project to keep the system maintainable?", "Where is it most beneficial to start a refactoring?", or "Does the quality of a third-party library / application meets our requirements?". Further, as Fenton and Pfleeger [FP97] write, *"measurement is needed at least for assessing the status of your projects"*.

To measure code, a large number of metrics is defined for all kinds of software systems, e.g., a well known set of metrics for object-oriented programs is defined by Chidamber and Kemerer [CK94] or discussed by Briand, Daly and Wüst [BDW98]. These metrics, as well as other metrics, are then used as changeability indicators [KKL01], to guide refactorings [SSL01], to try to estimate the quality of software projects [BBM96, BWDP00, BD02], to predict faulty classes [EMM01], to estimate the fault proneness in general [YSM02, FN01], or to determine those classes that need to be tested intensively [MSA+03]. Besides proposing new metrics, a large number of papers deal with the validation of metrics in general [EBGR99, FN99, Sch92, TCSD04, MSCM02], that is, finding correlations between metrics or finding the "best metric" for a specific task.

But, due to an ever changing software development process, new metrics need to be defined to measure modern software development projects in order

to support the development tasks discussed in the previous paragraph. In particular, being able to measure software projects beyond classical object-oriented and procedural systems is necessary. Modern software systems, such as Service Oriented Software Systems (SOSS), use different kinds of artifacts in the implementation of the system. In case of SOSS, a variety of XML dialects and files as well as conventional source code is used. In contrast to the previous use of XML files, these files are no longer mere configuration files. Instead, these XML documents, e.g., BPEL files [BPE05], implement important functionality of the system. Hence, it is necessary to be able to measure all documents and also to take the relations between different types of documents into account, when making an overall assessment.

The necessity to be able to consider relationships across different types of artifacts was the motivation for building a metrics framework on top of Magellan. Using Magellan and the embedded XQuery engine, it is immediately possible to implement and test new (cross-artifact) metrics.

Section 8.2 presents the prototypical metrics framework QScope and gives an overview of the architecture of QScope. Furthermore, work related to metrics frameworks is discussed and an evaluation of QScope as an extensible metrics framework is provided. Section 8.3 concludes this chapter with an evaluation of building a tool for code assessment upon Magellan.

8.2 QScope: an Extensible Metrics Framework

Currently, defining, implementing and testing metrics is labor-intensive. First, a meaningful metric has to be defined and after that, the metric needs to be implemented and measured for a number of real world projects in order to validate the metric. Though, it is practically impossible to support the first step since it usually requires creativity, the second step can be tool supported. Unfortunately, many metrics tools are not explicitly extensible or they are extensible, but are limited with respect to the set of different types of artifacts that can be taken into account. Hence, implementing metrics that need to analyze different types of artifacts to draw a conclusion is not yet explicitly supported by current tools.

The XQuery interface provided by Magellan is promising to make the development of new metrics easier. Using XQuery a uniform mechanism exists to calculate metrics for software systems where it is necessary to take multiple different types of artifacts into account. To evaluate the approach

a prototype called QScope was implemented. QScope offers the standard functionality expected from a metrics tool, that is, handling of metrics, visualization of a metric's result, aggregation of metric values, filtering metric values and exporting the results of calculated metrics. Further, an explicit mechanism to plug in new metrics is provided and a framework for developing and testing new metrics is also included. By relying on Magellan as the underlying platform, QScope is automatically open with respect to the types of artifacts that can be taken into account when measuring a software project.

8.2.1 Calculating Metrics using XQuery

In this subsection, the implementation of three different metrics using XQuery is shown. All queries use the XML representation of Java bytecode as generated by BAT$_2$XML. The first query calculates the basic metric Number of Methods. The second query is a cross-artifact metric that calculates the Number of Methods with Declaratively Specified Transaction Attributes and the third one analyzes a method's implementation to calculate the Lack of Cohesion in Methods.[1]

8.2.1.1 Number of Methods

In Listing 8.1 we see a complete example of a query to calculate the metric Number of Methods.

```
1 <metric> {
2  for $class in $db:prj−files/bat:∗[@name = $param]
3  let $fqn := $class/@name
4   return
5   <entity
6       package="{helper:packageName($fqn)}"
7       class="{helper:simpleName($fqn)}">
8           { fn:count(java:allMethods($class)) }
9    </entity>
10 } </metric>
```

Listing 8.1: XQuery for calculating the metric Number of Methods

[1] This section assumes that the reader has a basic understanding of XQuery (cf. Section 5.1).

The query first creates a new XML root element metric (Line 1). After that, all classes are selected where the class' name is matched by any item in the sequence $param (Line 2). The variable $param is an external variable that is initialized by QScope with the set of class names for which this metric is to be calculated. For each $class the fully-qualified name of the class is assigned to the local variable $fqn (Line 3) and a child element entity (Line 6) is created. The content of the entity element is the value of the metric (Line 8); i.e., the number of method declarations of the current class. The constructed XML elements (metric in Line 1 and entity in Line 6) are required to enable QScope to put a calculated value in relation to a resource; in this case, to a specific class.

The variable $db:prj-files (Line 2) as well as the functions helper: packageName (Line 6), helper:simpleName (Line 7) and java:allMethods (Line 8) are predefined by Magellan; the function fn:count is a standard XQuery function to count the number of elements.

Evaluating the query for the class demo.HelloBean shown in Listing 8.2 yields the result shown in Listing 8.3.

```
1 <bat:class name="demo.HelloBean">
2   <bat:inherits>
3     <bat:class name="java.lang.Object" />
4       <bat:interface name="javax.ejb.SessionBean" />
5   </bat:inherits>
6   <bat:method name="getText" ...>
7     <bat:signature>
8       <bat:returns type="java.lang.String" />
9     </bat:signature>
10    <bat:code>...</bat:code>
11  </bat:method>
12  <bat:method name="getValue" ...>
13    ... similar to getText
14  </bat:method>
15 </bat:class>
```

Listing 8.2: XML representation of demo.HelloBean

```
1 <metric>
2  <entity package="demo" class="HelloBean">
3   2
4  </entity>
5 </metric>
```

Listing 8.3: Result of calculating number of methods for demo.HelloBean

The result (Listing 8.3) of every XQuery which calculates a metric has to be an XML document with <metric> as the root element and one <entity> element per calculated value as shown in Listing 8.3, Line 2 and Line 3. Along with the entity element additional information (Line 2) is specified to enable switching from a metric's result to the code; e.g., in this case the value is the name of the class for which the metric was calculated.

8.2.1.2 Number of Methods with Declaratively Specified Transaction Attributes

To demonstrate how to use QScope to measure a project that makes use of different types of artifacts, we will see a metric that calculates the number of methods of an Enterprise JavaBean (EJB) [EJB03] class with declaratively specified transaction attributes. That is, those methods where the transaction attributes are specified in an XML deployment descriptor (see Listing 8.4) and not in source code. The motivation behind such a metric could be to measure *hidden complexity* in EJB projects. Though the transaction attributes are not specified in source code, it is in practice often not possible to implement an EJB without taking the transaction attributes into account. Hence, the complexity is hidden when analyzing the source code only.

Given the class shown in Listing 8.2 and the deployment descriptor (DD) shown in Listing 8.4, a function is defined (Listing 8.5) to determine those methods that have declaratively specified transaction attributes.

```
1 <ejb−jar>
2   <enterprise−beans><session>
3       <ejb−name>HelloBean</ejb−name>...
4       <ejb−class>demo.HelloBean</ejb−class>...
5   </session></enterprise−beans>
6   <assembly−descriptor><container−transaction>
7       <method>
8         <ejb−name>HelloBean</ejb−name>
9         <method−name>getText</method−name>
10      </method>...
11  </container−transaction></assembly−descriptor>
12 </ejb−jar>
```

Listing 8.4: Abbreviated EJB deployment descriptor for demo.HelloBean

```
1 declare function ejb:methodsWithDeclTransAttrs($class ) {
2   let $ejb-name := $ejb:ejb-jars/enterprise-beans/*[./ejb-class/text() =
        $class/@name]/ejb-name
3   let $method-names := $ejb:ejb-jars//container-transaction/*[./ejb-name =
        $ejb-name]/method-name/text()
4   return $class/bat:method[@name = $method-names]
5 };
```

Listing 8.5: Methods with declaratively specified transaction attributes

The function first determines the ejb-name of the given class (Line 2) by searching for the class' name in the EJB deployment descriptor (Line 4 in Listing 8.4). Having the ejb-name, the query then (Line 3) selects the names of all methods with declaratively specified container transactions (Line 9 in Listing 8.4). At last, the methods of the class with matching names are selected (Line 4).

Given the function defined in Listing 8.5, calculating the metric for a specific class is straightforward. We pass the class (by means of a variable $class) to the function and count the number of (method) nodes returned:

```
1 {fn:count(ejb:methodsWithDeclTransAttrs($class))}
```

The complete result of evaluating the query is then:

```
1 <metric><entity class="HelloBean" package="demo">1</entity></metric>
```

8.2.1.3 Lack of Cohesion in Methods

Even for metrics that analyze a method's implementation, a query definition close to the mathematical definition is possible. For example, LCOM measures the cohesion of a class, i.e. how methods and variables of a class relate to each other. The precise definition is shown in Figure 8.1: A is the class for which we want to calculate the lack of cohesion, k is the number of fields declared by A, n is the number of methods of A, and v_i is one specific field. Hence, $P(v_i)$ is the percentage of methods that access the field v_i.

The query to calculate the metric is shown in Listing 8.6. The for loop in Line 1 iterates over all classes passed to the query. For a specific class $A, Line 2 to 5 determine the set of fields and methods and the sets' sizes. After that, Line 7 to 18 calculate the metric. In Line 7 and 8 the special case that the class defines either no fields or no methods is handled. In all other cases, the metric is computed in the lines 10 to 17 as specified in Figure 8.1.

$$LCOM(A) = 100\% - (\frac{1}{k}\sum_{j=1}^{k} P(v_j)),$$
$$\text{where } P(v_i) = \frac{\text{Number of methods accessing } v_i}{n} 100\%$$

Figure 8.1: The Lack of Cohesion in Methods (LCOM) metric

```
1 for $A in $db:prj−files/bat:class[@name=$param]
2 let $methods := java:all−methods($A)
3 let $n := fn:count($methods)
4 let $fields := java:all−declared−fields($A)
5 let $k := fn:count($fields)
6 return
7   if ($k = 0) then 0
8   else if ($n = 0) then 100
9   else (
10    (1−
11      fn:sum(
12        for $v in $fields
13        return fn:count(
14            java:accessors($v, $methods)
15          ) div $n
16      ) div $k
17    ) ∗ 100
18  )
```

Listing 8.6: XQuery for calculating the Lack of Cohesion in Methods

Listing 8.7 shows the function to determine those methods, among a set of given methods ($methods), that access a specific field (used in Listing 8.6, Line 14). The query in Listing 8.7 returns a method m, if the set of put and get instructions that access the given field $v is not empty — put and get represent the field write and read access instructions.

```
1 declare function java:accessors($v, $methods) as element()∗ {
2  $methods[not(empty(.//(bat:put | bat:get)
3            [./@declaringClassName=$v/../@name
4              and ./@fieldName=$v/@name]))]
5 };
```

Listing 8.7: XQuery to get the methods accessing a specific field

8.2.2 Architecture

QScope is comprised of the four building blocks shown in Figure 8.2.

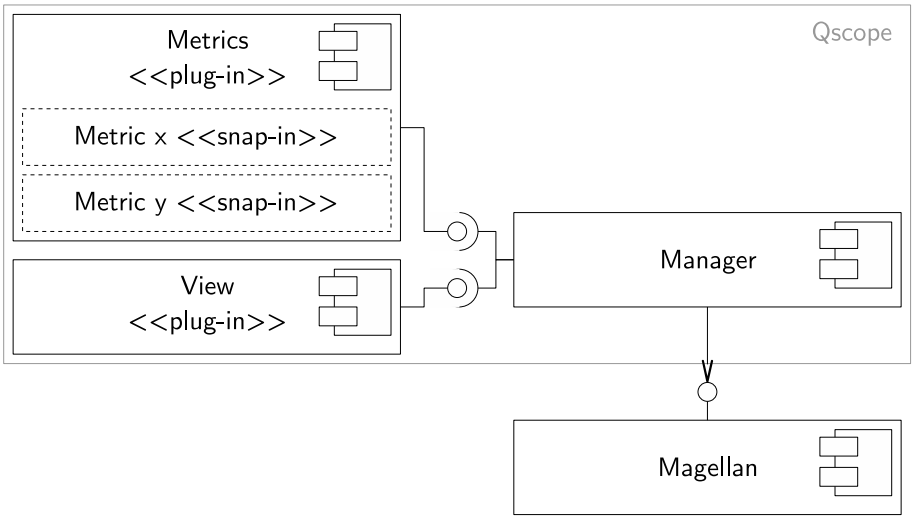

Figure 8.2: Architectural overview of QScope

Magellan
QScope builds upon Magellan and uses the provided XQuery interface for the calculation of metrics.

Manager
The central component of QScope is the *Manager*. It is written as an Eclipse plug-in and serves as a connector between (a) the metrics plug-ins, (b) the views, and (c) Magellan. The interaction between the Manager and the other components is discussed next. To achieve extensibility, the Manager plug-in provides two Eclipse extension points that enable developers to extend QScope with new views and metrics, respectively.

Metrics
Each metric is defined as a so called *snap-in*. Snap-ins contain the

metric's declaration together with an XML descriptor for the metric. The latter includes the meta-information of the metric: the name of the metric, a short description, documentation, and a reference to the XQuery file. Furthermore, each metric descriptor contains a scope element that enables to limit the metric evaluation to certain entities, i.e., packages, classes, interfaces, methods or fields. For instance, the scope of the metric Depth of Inheritance Tree (DIT) is class. Hence, it is not possible to calculate the metric for a method or a field, but it is possible to calculate the DIT for a package. In this case, the metric is calculated for all classes within the selected package and the result is aggregated according to the user-defined aggregation setting (e.g., the mean value or deviation).

To register new metrics, developers create a metrics plug-in that aggregates metric snap-ins that belong together, e.g., metrics for object-oriented programs or metrics for J2EE application. The metrics plug-in then uses the extension point defined by the Manager. When QScope is loaded by Eclipse the extension point is used to discover all metrics plug-ins and to automatically register the contained snap-ins.

Views

Users can interact with QScope using views, i.e., select and visualize metrics. Currently, two different views are available: a spreadsheet like *TableView* (bottom part of Figure 8.3) and a graphical *ChartView* (central part of Figure 8.3). Both views contain a configuration pane that is used to select the metrics of interest. Furthermore, one can setup the aggregation level for the metrics to a higher-level entity and define filters.

Depending on the concrete view, it is also possible to define the sorting order (for the TableView) or to specify a diagram type and turning the diagram's legend on and off (ChartView).

The TableView facilitates to see the exact results of metrics, as well as to quickly find the minimum or maximum value by ordering the results correspondingly. Furthermore, the results can be exported to an XML file. The ChartView can be used to visualize the results with various diagram types. Currently, six diagram types are implemented: bar chart, bubble chart (seen in Figure 8.3), histogram, pie chart, scatter plot, and waterfall chart. The graphical view allows to quickly perceive the distribution of metric values as well as mavericks. For example, in

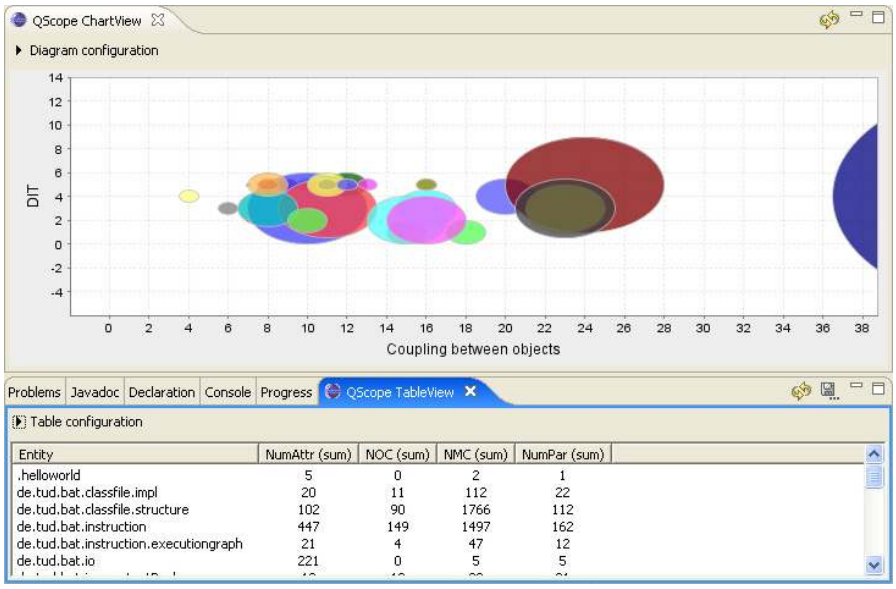

Figure 8.3: Screenshot of QScope

Figure 8.3 the ChartView shows a bubble chart which visualizes three metrics: Coupling Between Objects (CBO), the Depth of Inheritance Tree (DIT), and the Lines of Code (LoC). Each bubble represents one class in the system where the center of the bubble is determined by the CBO (x-axis) and the DIT (y-axis) values of a class; the diameter of the bubble corresponds to the number of Lines of Code.

8.2.3 Using QScope

To calculate metrics, the user selects the entities for which the metrics should be calculated, e.g., the whole system, some packages, or some classes. Then the metrics of interest have to be selected and the aggregation level at which the metrics should be calculated. This information is passed to the Manager component. The Manager then loads the metrics' queries and evaluates them using Magellan. For each query Magellan returns an XML document containing the results. The Manager then forwards the results to the view. Finally, the view visualizes the results.

8.2.4 Extending QScope

To plug-in a new metric in QScope it is necessary to:

1. To develop a new metric snap-in that consists of the query to calculate the metric and the XML descriptor with the necessary meta-information. Both artifacts are saved in the metrics directory and are then available.

2. To optionally test the newly developed metric. Metrics for modern programming languages have to cope with a number of special cases. For instance, sometimes one has to consider the inheritance hierarchy and for some metrics interfaces and classes are handled differently. To ensure that the metric calculates the right results in all cases, testing becomes a major concern. QScope offers a test framework for unit testing queries to provide developers an appropriate environment for assessing a metric's correctness. A prerequisite for effective unit testing of metrics is a test project that contains XML documents that reflect all cases that should be tested. One either has to create such a project or use an existing one, e.g., one of the test projects shipped with QScope. The only requirement is that the project has to be a plug-in project and that Magellan is enabled.

   After implementing the metric and defining the meta-information, developers have to manually specify the expected values for the entities they are interested in. These values correspond to the expected values in a JUnit test method and are stored in an extra XML file. Then, the developer creates a metric unit test class by inheriting from the provided class MetricTestCase and specifies the metric descriptor as well as the XML file with the expected values. The metric tests are then executed as an Eclipse plug-in unit test.

3. After the snap-in has been developed and tested, developers can add it to an existing metrics plug-in or create a new one. The plug-in is the deployable unit and enables other users to integrate the metric inside their IDE.

8.2.5 Evaluation

To evaluate QScope and the applicability of XQuery for defining metrics, a set of 18 metrics was implemented. Most of them were developed using the definitions given in [YSM02] and are shown in Table 8.1. In general, the implementation of the metrics was straightforward and, as discussed in section 8.2.1, even metrics as complex as Lack of Cohesion in Methods can be

Name	Description
1. Coupling Between Objects (CBO)	the number of other classes to which a class is coupled [YSM02]
2. Coupling within an inheritance hierarchy (CBO_{in})	CBO that are in a subclass-superclass relationship [YSM02]
3. Coupling across inheritance hierarchies (CBO_{out})	CBO_{out} = CBO-CBO_{in} [YSM02]
4. Depth of Inheritance Tree (DIT)	the longest path from a given class to a root class / interface within the inheritance hierarchy [YSM02]
5. DIT_{cls}	DIT without considering interfaces
6. Lack of Cohesion (LCOM)	*see discussion*
7. Number of Ancestor Interfaces (NAI)	number of directly and indirectly inherited interfaces
8. Number of Ancestors (NoA)	number of distinct supertypes
9. Number of Fields (NoF)	number of fields of a given class
10. Number of Children (NOC)	number of immediate subtypes [YSM02]
11. Number of Classes (NCP) / 12. Interfaces in a Package (NIP)	number of classes / interfaces in a package
13. Number of Methods (NMC) / 14. Constructors per Class (NCC)	number of methods / constructors per class [YSM02]
15. Number of Parents (NoP)	number of immediate supertypes
16. Response For a Class (RFC)	number of distinct methods called by a given class, but where the methods are defined in other classes [YSM02]
17. RFC_{in}	RFC, but limited to those methods defined in a superclass [YSM02]
18. RFC_{out}	RFC_{out} = RFC - RFC_{in} [YSM02]

Table 8.1: Metrics implemented in QScope

implemented close to the mathematical definition.

The performance of QScope is evaluated based on calculating all 18 metrics for the LimeWire[Lim06] project. LimeWire is an open source client on the Gnutella network and consists of 1387 classes and interfaces. For each metric, the runtime was measured twice: First, using the open source SaxonB [Kay05a] XQuery processor and, second, using the commercial variant SaxonSA [Kay05b]. Since both variants use the same interfaces exchanging the open-source against the commercial variant just required to replace the corresponding jar archives.

The results are depicted in Figure 8.4; please note that the Y-axis uses a logarithmic scale (log 10). In case of the measurements taken using SaxonSA, additionally to exchanging the XQuery processor, slow queries were rewritten to make use of an indexing function only available with SaxonSA. Basically, the created index enables a direct jump to an XML class element given a class' name. Using the index traversing the type hierarchy is extremely fast as it is no longer necessary to search the entire database for a super-/subclass. After figuring out how to improve the performance, the rewrite took in average 10 minutes per query. The depicted execution times include the time required for generating the index. Besides using the indexing function, no explicit use of any further features available with SaxonSA was made; in particular no use of the processor's feature to optimize a query based on the XML schema was made. Using this feature would require an XML schema definition for the entire database which is currently not available.

The performance measurements were made on a Windows XP Pentium M 1,6 GHz notebook with 2 GB of main memory. As the performance figures show, using QScope and SaxonSA with manually created indexes leads to competitive runtime performance in all cases; the slowest query takes about 12 seconds when evaluated for the whole project. In average evaluating a query requires 3.94 seconds. However, using the open source SaxonB query processor half of the queries are too slow for practical purposes; in particular those queries that need to traverse the inheritance hierarchy. Hence, using QScope with the open source SaxonB XQuery processor is only possible for small projects; measuring larger projects with several hundreds of classes requires to use the commercial SaxonSA XQuery processor.

8.2.6 Related Work

A large number of tools for calculating metrics exists, e.g., Borland Together Control Center [Tog05], the Code Analysis Plug-in (CAP) [CAP05],

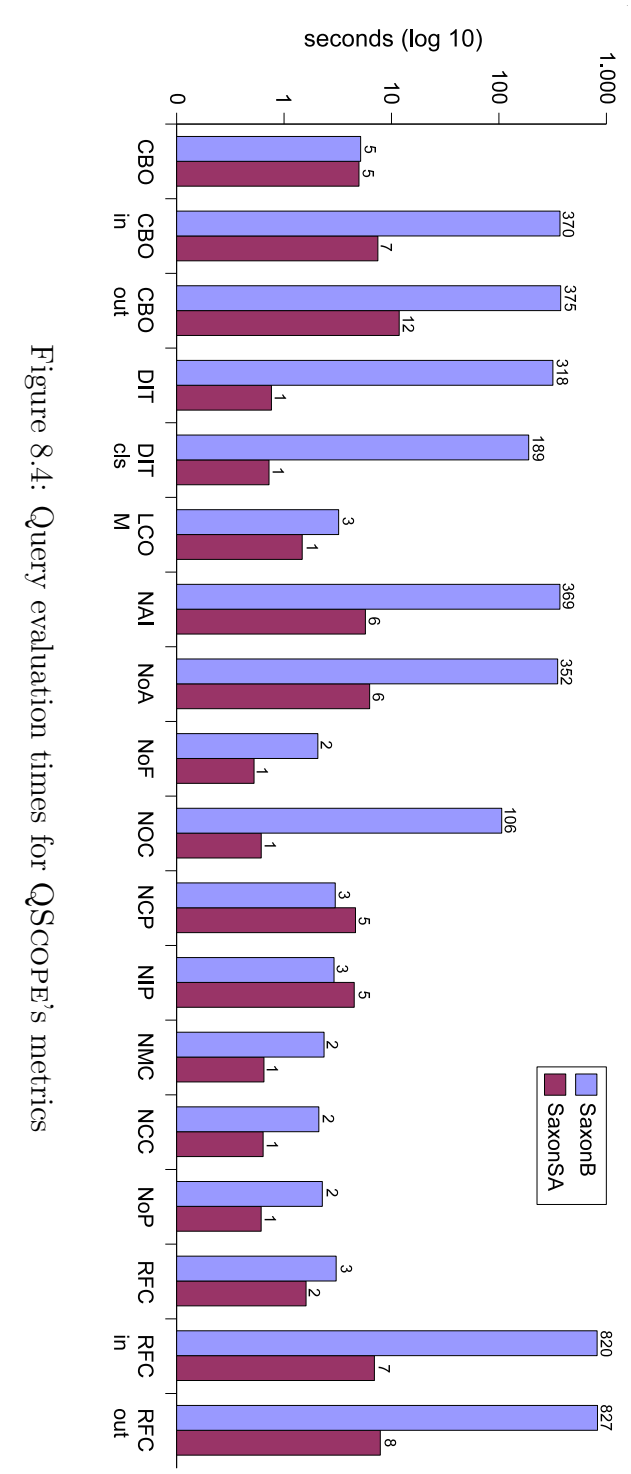

Figure 8.4: Query evaluation times for QScope's metrics

EclipsePro Audit [Ecl05], JMetric [JMe99], Metrics [Met05] and SDMetrics [SDM06]. However, most of these tools either provide only a fixed set of built-in metrics or enable the user to extend the built-in set of metrics, but do require to implement the metrics in a standard programming language, such as, Java or C#. Furthermore, most tools only enable to write queries for applications written in a programming language directly supported by the tool. Taking other documents into consideration, e.g., XML deployment descriptors to setup the runtime environments for components, is non-trivial. QSCOPE is not limited to particular types of artifacts, because it uses the generic query language XQuery for calculating metrics and is built upon MAGELLAN.

A metrics tool that targets user extensibility is OOMeter [ARK05]. OOMeter extracts elementary information about Java and C# source code as well as XMI files and stores the information in a repository. The repository is then used for the calculation of higher-level metrics, e.g., coupling and cohesion metrics. OOMeter can be extended by implementing new metrics as Java classes that implement a specific interface and that operate on the repository. Hence, compared with QSCOPE the set of new metrics that can be defined using OOMeter is limited to metrics that can be defined on top of the elementary information stored in the repository, while QSCOPE provides full access to all documents of the application. When compared to OOMeter, QSCOPE makes the development of new metrics easier by providing a set of predefined, reusable queries for extracting common information instead of preprocessing the source files.

The Moose [DT03] reengineering environment uses a similar approach as OOMeter: the source code is analyzed and the extracted information is stored in a repository with the FAMIX meta-model, which was designed with respect to the features commonly found in object-oriented programming languages. Moose's meta-model is extensible and provides support for language specific features. Even though Moose provides support for the definition and calculation of metrics by means of an API, Moose does not provide a declarative query language for the definition of new metrics.

Lewerentz and Simon [LS97, LS98] describe the metrics tool Crocodile, which also uses a declarative SQL-like query language to calculate metrics. Further, similar to QSCOPE Crocodile is also integrated in an IDE and executes the queries against the program database that is maintained by the IDE. However, compared to QSCOPE Crocodile is limited in several ways. First, the database only contains information about the high-level elements of object-oriented programs, such as, classes, methods and fields. Second, the

query language's capabilities were designed with respect to the elements that are possibly contained in the database and is not general purpose. Hence, even though Crocodile is language independent, it cannot be used to calculate metrics beyond metrics for object-oriented source code.

Marinescu et al. [MMG05] identify a set of five key mechanisms that are the building blocks for the implementation of structural analyses; e.g., for metrics. The key mechanisms are: navigation (e.g., given a class go to the declared methods), filtering (e.g., get only the methods with return type integer), selection (e.g., select the name of a method), set arithmetic (e.g., compute the union of two sets of selected methods) and property computing analyses (e.g., count the number of items in a set). Since QScope uses XQuery as the language for the implementation of the metrics the identified key mechanisms are well supported; XQuery already provides built-in support for all five mechanisms. However, compared with the prototype language SAIL [MMG05], a standard language is used and it is not necessary to mix imperative programming and a query mechanism to calculate the metrics as in case of SAIL; solely using XQuery proved to be sufficient in all cases.

8.3 Conclusions

In this chapter, the metrics platform QScope was presented. The openness of QScope is directly due to the openness of the underlying platform Magellan and enables to extend the set of artifacts that can be taken into account when running queries against the code base. To be extensible QScope defines Eclipse extension points to plug-in new metrics written in XQuery.

To assess if it is possible to use XQuery for calculating metrics for larger projects 18 well known metrics were implemented and the evaluation times for a reasonable sized project were measured. The result of the evaluation shows that QScope is a good platform for developing and prototyping new metrics and that using QScope/Magellan/XQuery facilitates the definition of complex metrics. As the performance figures have shown, using a commercial XQuery processor the execution times of the queries are fast enough for projects with several hundred classes and other artifacts.

We have also seen that XQuery can be used to calculate metrics for software systems where relevant information is defined across different types of artifacts. In the given example, the metric analyses Java code in conjunction

with XML based deployment descriptors.

The following list summarizes the advantages of building a metrics framework on top of MAGELLAN:

- For all types of artifacts for which an XML mapping is available it is immediately possible to implement metrics. No functionality is required to build up and maintain a database which stores the current software's model. Hence, by using MAGELLAN it is possible to focus on the *business functionality of a metrics framework*: implementing and visualizing metrics, providing functionality to filter and aggregate metric values and to export results.

- As soon as the set of artifacts with an XML mapping is extended, it is immediately possible to write (new) metrics which take the information into account. Since other tools, such as software comprehension tools (cf. Section 7.3) and tools for checking structural properties (cf. Section 6.3), can also use the same representation, adding support for new types of artifacts is very beneficial and more likely to happen.

- Since MAGELLAN's meta-model of the XML database is extremely lightweight and facilitates the integration of all types of documents, QSCOPE is not restricted to a particular programming language or paradigm, such as, object-oriented, functional, or procedural programming.

- The (X)query interface provided by MAGELLAN enables a concise definition of new metrics and enables the rapid development of a metrics suite.

- When implementing new metrics performance issues are not a major concern — when using a commercial grade XQuery processor. That is, it is possible to write a Query as close to the mathematical definition as possible and it is not necessary to do extensive manual performance tuning. Nevertheless, since XQuery is a rather new standard the performance of the query engines will certainly improve and will probably diminish the need for manual optimizations at all.

- The seamless integration with horizontal tools that metrics tools have to provide, as argued by Auer et al. [AGB03], is readily available when using MAGELLAN. Eclipse is (becoming) a universal tool platform

which is going to support the whole life cycle of an application — from design over implementation to maintenance and MAGELLAN is tightly integrated into Eclipse.

Chapter 9

Advanced Type Systems

This chapter shares some material with: *Incremental Confined Types Analysis* [EKMS06]

9.1 Introduction

Type systems represent formal methods to prove the absence of certain (erroneous) program behaviors by calculating a kind of static approximation of the run-time behavior [Pie02]. Being static, the approximations are conversative and, hence, sometimes otherwise valid programs will be rejected. However, when using a type system it is in particular possible to detect errors early and to pinpoint to the source of the error. For example, type systems for object-oriented languages prevent the sending of messages to objects that do not have corresponding methods. With type systems these errors can be detected statically and, hence, it can be guaranteed that corresponding runtime errors will never occur [Bru02].

The advantages most often outweigh the disadvantage of rejected programs and the additional effort required for making the type information explicit in the source code. Other advantages of type systems mentioned in the literature are: they help to name and organize concepts [Mit03], they serve program comprehension when reading programs [Pie02], or they are used by compilers to generate more efficient code [Pie02].

The benefits of type systems have been recognized by many researchers and have led to the development of various advanced type systems. For example, type systems to prevent deadlocks and data races [BLR02, BR01], to control aliasing [NVP98, VB01], or to support non-null types [FL03] to

name just a few.

Unfortunately, the use of these advanced type systems is not widespread. Most implementations are proofs of concept and fall short with respect to the integration with standard software development tools and processes. This lack of integration was one motivation for the work on implementing advanced type systems using MAGELLAN. Since MAGELLAN is integrated with the incremental build process of Eclipse, integration issues are no longer a concern. Further, many tool adoption issues [BJL+03, FES03] are also avoided. The user will, after activation, perceive no difference between the analyses carried out by the standard Java compiler and the activated analyses. This flattens the learning curve as it is not necessary to learn how to use different tools. Additionally, since we (re)use the standard Eclipse views to visualize errors no user interface related issues arise.

A second motivation was to test how well suited Magellan is for implementing *pluggable type systems* [Bra04]. Pluggable type systems are optional type systems that are selected based on a project's needs and which are plugged in to the compilation process. Though, optional type system are neither syntactically nor semantically required they can still provide most of the benefits of mandatory, compiler built-in type systems listed above. The only exception is that optional type systems are not used by the compiler as a source of information for optimizations. Reasons for making a type system pluggable instead of mandatory are:

- It is not possible to integrate every conceivable type system into a programming language as this would make the programming language overly complex and restrictive. As stated by Nobel et al. [NVP98]: *"The success and acceptance of a type system in practice depends on the extent to which it supports or constraints idiomatic programming style."*

- Enforcing specific typing rules is not always advantageous. As written in [NVP98] *"... it is demonstrably the case that these [aliasing] problems do not manifest themselves in the vast majority of programs"*. Hence, forcing the developer to add the necessary type annotations (to control aliasing) to every program would waste time and effort in most cases. Using a type system, e.g., for alias control is only useful for security related parts of an application.

- When developing new type systems it is first necessary to collect experiences and to evaluate different variants. For example, as written

> by Zaho et al. [ZPV03]: *"... before settling on one particular notion of confinement and incorporating that in a new language design, it is necessary to get first-hand experience with the benefits and costs of developing large software with these new constructs."*

A prerequisite for pluggable type systems is a generic mechanism at the language level that facilitates adding type annotations to the source code. Though, the metadata facilities of current languages, such as e.g., Java annotations and C# attributes, were not designed for pluggable type systems, they are at least sufficient for the type systems considered in this thesis.[1] Using metadata facilities for adding type annotations has the advantage that compatibility with existing tools is guaranteed. The major disadvantage of using the metadata facilities is that the compiler is not aware of the additional type system and, hence, typing errors are only reported if the optional type system's analyses are carried out additionally to those of the compiler. That is, source code that can successfully be compiled is not guaranteed to be error free with respect to the optional type system. However, this is not considered to be a major problem in the context of open static analysis platforms. By sharing the project's analysis configuration a project leader can enforce that specific analyses are activated.

In the following section, an implementation of confined types on top of Magellan is presented. The type system's analyses are implemented in Java and in Prolog to enable a comparison of both approaches. For the type annotations, both implementations rely on the same set of standard Java 5 annotations. Section 9.3 concludes this chapter with a discussion of realizing optional type systems on top of Magellan. Further, it is reasoned about implementing analyses in Java and in Prolog.

9.2 Confined Types as an Optional Type System

9.2.1 Introduction

Aliasing is pervasive in object-oriented programming and can cause many kinds of problems, if unintended. However, in the majority of cases aliasing is benign and is not a source of programming errors [NVP98]. Neverthe-

[1] In the following, the term *annotations* is used to refer to Java annotations and not to refer to type annotations.

less, aliasing hampers modular reasoning, as it is hard to analyze the effect of updating an object when it is unknown which other objects also keep references.

Besides making program comprehension harder, unintended aliasing can also lead to subtle errors. For example, an Enterprise Java Beans (EJB) container needs to have full control over the whole life cycle of all its beans for the correct operation of its services, such as, pooling and persistence. To ensure the necessary control, an EJB is not allowed to directly pass its self-reference (this) to other beans to avoid creating aliases that are not controlled by the container. The following sequence, for example, might cause an erroneous behavior of the application: An entity bean passes its this reference to a session bean, then the container's persistence service persists the entity bean and sets all references to it to null to make it garbage collectable. But, since the session bean still holds a reference to the entity bean the garbage collection will fail; the container's memory management functionality is circumvented. Further, since the state of the entity bean is no longer synchronized with the database by the container the application's result is unpredictable when the bean is subsequently used.

Besides being a source of programming errors that can be detected when testing an application, unintended aliasing can also lead to security errors, which are hard to detect using standard development techniques. For example, when a reference to an object is passed to another object and, hence, an alias is created for the first object, then the alias can later on be used to update the first object in an unanticipated manner. In [VB01] a security breach caused by a reference leaking bug in the JDK 1.1 is discussed (shown in Listing 9.1).

```
1 public class Class {
2   private Identity[] signers;
3   public Identity[] getSigners() {
4     return signers;
5   }
6 }
```

Listing 9.1: JDK1.1 implementation of Class.getSigners()

In the JDK's implementation, each instance of a Java Class object holds an array of signers (line 2) that represents the principals under which the class acts. The problem is that the getSigners method returns a reference to the original signers array (line 4). Hence, the attackers can freely update the

signatures based on their needs.

To solve the problems related to the creation of unintended aliases, means are needed to enforce that important data structures cannot escape the scope of a well-defined protection domain. For example, to assure that the reference to the original signers array does not escape the declaring class. A naive solution to avoid the breach shown in Listing 9.1 is a programming style that encourages the developers of classes with sensitive information to return a reference to a copy of the sensitive data, in our case a copy of the signers array. But, programming styles cannot be enforced. Using an appropriate type system, it is possible to ensure that none of the key data structures used in code signing escape the scope of their defining package.

To restrict aliasing to certain protection domains, i.e., to prevent leaks of sensitive object references, Vitek and Bokowski [VB01] propose confined types.

Confinement ensures that objects of a confined type can only be accessed within a certain protection domain. A type is confined to a domain if all references to objects of that type originate from within the domain. Code outside the protection domain is never allowed to manipulate confined objects directly. For this purpose, types whose instances should not leave their defining package are marked as confined. In contrast to existing access control mechanisms in Java (such as the Java private keyword), confinement constrains access to object references rather than classes.

In contrast to the original proposal Java 5 annotations are used for confined types instead of defining new modifiers. The modifiers: confined and anon used in [VB01] are replaced with the annotation types: @confined and @anon.

Listing 9.2 shows, how the code from Listing 9.1 can be rewritten using confined types. The annotation @confined is used with a class, whose objects should be confined to the containing package. In Listing 9.2 annotating SecureIdentity as @confined (line 3) enforces references to SecureIdentity objects to be confined to the package java.security. Thus, code outside this package can never access instances of type SecureIdentity. Renaming the old Identity class to SecureIdentity and introducing a new Identity class (line 4 – 8) preserves the functionality of the original interface.

```
1 package java.security;
2   abstract class AbstractIdentity { @anon equals(){...};  }
3   @confined  class SecureIdentity extends AbstractIdentity { ... }
4   public class Identity {
5     SecureIdentity target;
```

```
6     Identity(SecureIdentity t) { target = t; }
7     ... // public operations on identities;
8   }
9   public class Class {
10    private SecureIdentity[] signers;
11    public Identity[] getSigners( ) {
12      Identity[] pub = new Identity[signers.length];
13      for (int i = 0; i < signers.length; i++)
14        pub[i] = new Identity(signers[i]);
15      return pub;
16    }
17  }
```

Listing 9.2: Class.getSigners() using Confined Types

The @anon annotation enables confined types to safely use methods from unconfined types. Methods that do not reveal the current object's identity are marked as *anonymous* by annotating them with @anon. This annotation serves two purposes: to document the method's intention and to make this property machine checkable. In Listing 9.2, the method equals in line 2 is marked with @anon to show that it never reveals the current instance's identity (this). Therefore, SecureIdentity can safely extend AbstractIdentity and call equals on this.

The constraints in Table 9.1 and 9.2 define the semantics of @confined and @anon. Constraints in Table 9.1 restrict class and interface declarations (*C1*, *C2*), prevent widening (*C3*), hidden widening (*C4*, *C5*), and transfers from inside (*C6*) and outside (*C7*, *C8*) the protection domain. The rules defined in Table 9.2 constrain the usage of the self-reference this in method implementations, so that this is not revealed to code outside the method.

The constraints *C2'* and *A2'* defined in Table 9.1 and 9.2 are slightly modified when compared to the original definition of *C2* [VB01]: "Subtypes of a confined type must be confined as well." and *A2*: "Anonymity of methods and constructors must be preserved in subtypes." These modifications reduce the number of classes that need to be type checked in case of an incremental change. These modifications do not affect the semantics of confined types: A program satisfies all the constraints from Table 9.1 and Table 9.2 if and only if it satisfies them with *C2'* and *A2'* replaced by *C2* and *A2*.

Using confined types as an extension to the Java type system, the programming style of returning only copies of sensitive data can be supported in such a way that once a type is marked as @confined, the safety of the program

C1	A confined class or interface must not be declared public and must not belong to the unnamed global package.
C2'	If a direct super-type of a type t is confined, t must be confined as well.
C3	Widening of references from a confined type to an unconfined type is forbidden in assignments, method call arguments, return statements, and explicit casts.
C4	Methods invoked on a confined object must either be non-native methods defined in a confined class or be anonymous methods.
C5	Constructors called from the constructor of a confined class must either be defined by a confined class or be anonymous constructors.
C6	Subtypes of java.lang.Throwable and java.lang.Thread may not be confined.
C7	The declared type of public and protected fields in unconfined types may not be confined.
C8	The return type of public and protected methods in unconfined types may not be confined.

Table 9.1: Constraints for confined types

A1	The reference this can only be used for accessing fields and calling anonymous methods of the current instance or for object reference comparisons.
A2'	If a method m directly overrides an anonymous method, m must be anonymous as well.
A3	Constructors called from an anonymous constructor must be anonymous.
A4	Native methods may not be declared anonymous.

Table 9.2: Constraints for anonymous methods

with respect to avoiding unintended reference leaking can be guaranteed.

Figure 9.1 shows a violation of a confined types rule. To report the error the standard Eclipse Problem View is used and, hence, the user cannot distinguish this error report from other compiler generated error reports.

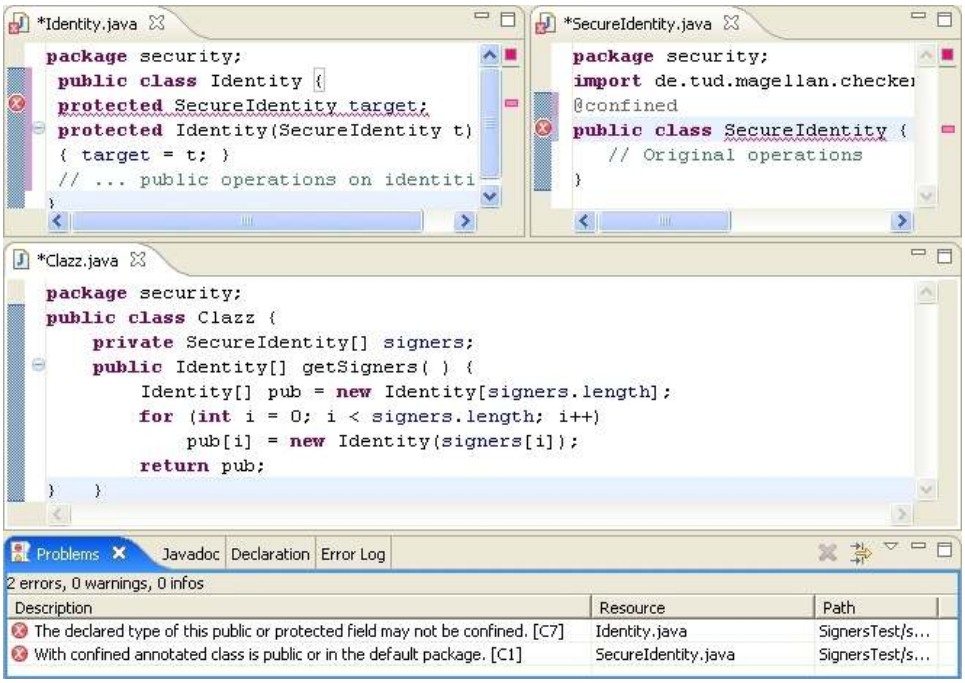

Figure 9.1: Screenshot of Eclipse when using confined types

9.2.2 Implementation

As stated in [VB01], checking the confinement rules is modular in the sense that each class can be analyzed separately. However, in addition to modularity and dynamic loading, it is also necessary to support continuous checking of confinement constraints during a programming task. In such a setting, checking all constraints on all classes after every change is obviously prohibitive in terms of incremental build performance. However, determining which classes have to be reanalyzed after a set of arbitrary changes to the project's source code is non-trivial. For an example of how a small change can impact the confinement rules at seemingly unrelated locations consider Listing 9.3.

```
1 package x;
2   public class X1 {
```

```
3     @anon public void m() { /* ... */ }
4   }
5   public class X2 {
6     public void m() { /* ... */ }
7   }
8
9  package y;
10  public class Y extends X1 { } /* change: ... extends X2 */
11
12 package z;
13  @confined class Z extends Y { /* ... */ }
14  class W {
15    public void foo() {
16      Z z = new Z();
17      z.m(); /* will violate C4 after change */
18  }
19 }
```

Listing 9.3: Indirect violation of confinement constraints

The example consists of Java classes in three different packages. Class W calls a method m on a confined class Z. *C4* is satisfied because Z inherits m from class X1 where it is declared anonymous. Now, let us assume that Y is changed to inherit from X2 instead of X1. Since X2 does not declare m as anonymous, the method call in line 17 now violates constraint *C4*. Hence, a change in package y (which does not contain any confined or anonymous declarations) yields a confinement error in a class in package z that is neither a subtype nor a supertype of the changed class Y.

The example shows that when a class changes, it is not sufficient to only check classes in the same package / protection domain or all super-types and subtypes of the changed class.

9.2.2.1 Java Based

In the following, a systematic approach to develop an incremental algorithm for checking the confinement rules is presented.

The checking algorithm is designed in two steps. First, given a list of classes that have been changed a set of classes is identified that must be reanalyzed to discover any new constraint violation and to remove any error message for constraints that are no longer violated. Next, the constraint rules are checked for all classes returned by the first step. Whenever a check fails

an error report is created and presented to the user (see Figure 9.1). Hence, after editing a source file, the developer is immediately informed about all current constraint violations.

All constraints from Table 9.1 and Table 9.2 are regarded as predicates over classes, respectively over methods. For any class c, $C_i(c)$ is true, if and only if c satisfies C_i. For any method m, $A_i(m)$ is true only if m satisfies the constraint A_i. Each predicate can be evaluated on its own since the definitions of the constraints do not depend on each other. For example, for a class c to satisfy constraint *C4* it suffices that methods called on confined types within c are declared as anonymous. Whether these methods, in turn, satisfy the constraints for anonymous methods is irrelevant for *C4*, though. The reason is that error messages are directly related to predicates that are not fulfilled. Violations of the constraints for anonymous methods will be displayed as separate errors when analyzing the respective methods.

Now the problem can be stated as follows: Given a program, the set of changed classes, and the predicate values for all classes and methods, update the predicate values such that they reflect the program changes. This update process should be correct in the sense that it produces the same results as a whole-program analysis.

In the following, for each constraint the information it depends on is determined; the constraint has to be reevaluated only if this information changes.

First, the rules for anonymous methods as defined in Table 9.2 are investigated.

- $A1(m)$ depends on the anonymous attribute of all methods called on this inside m. These methods have been declared either in m's class or in a supertype of the latter. Hence, for any changed class c, $A1(m)$ must be reevaluated for any m in c or any of its subtypes.

- $A2'(m)$ depends on the anonymous attribute of the method overridden by m. Since such a method must be declared in a supertype of m's class, the same invalidation strategy as for $A1$ applies.

- Since calls to constructors from within a constructor can be seen as a special kind of method calls on this, we can treat *A3* in the same way as *A1*.

- *A4* does not depend on any non-local information. Thus, it suffices to reevaluate *A4* on all methods of a changed class.

This leads to the following incremental algorithm for checking the constraints from Table 9.2. Whenever a type t changes, the constraints *A1–A3* have to be reevaluated on all subtypes of t (including t itself). Constraint *A4* only has to be reevaluated for types that have been changed.

Next, the constraints in Table 9.1 are analyzed in the same way.

- $C1(c)$ only depends on information from the class c. Thus, for every c, which has changed, $C1(c)$ must be reevaluated.

- $C2'(c)$ depends on the confined attribute of all direct supertypes of c. Thus, we have to reevaluate $C2'(c)$ for any c that is a direct subtype of a changed class c'.

- $C3(c)$ depends on the confined attribute of the types used in widenings inside one of c's methods. The value of $C3(c)$ can change only if either c is changed (so that the list of widenings performed inside c has changed) or if the confined attribute of a type t that is used in a widening changes. For each such t, the following holds: t has been confined at some point (i.e., before or after the change), hence, t is defined within the same package as c. Therefore, for each class c whose confined attribute has changed $C3$ needs to be reevaluated for any class in the same package as a class c.

- $C4(c)$ depends on method calls in c where the static type of the receiver is confined. More specifically, it depends on the confined attribute of the method's declaring type and the method's anonymous attribute.

  Since the static receiver type is confined, it must be in the same package as the class that contains the method call. Thus, whenever the confined attribute of a type t changes, $C4(c)$ must be reevaluated for any class c in the same package as t to recheck all relevant method calls on t.

  Additionally, *C4* has to be reevaluated when the anonymous attribute of the called method changes. This can happen indirectly as shown in the example from Listing 9.3. Thus, whenever a type t is changed all classes have to be determined that call a method on a confined subtype t' of t. Since a confined type can only be package visible, such a class must be in the same package as t'. For every confined subclass t' of t we check $C4(c)$ for all classes c in t''s package.

- The constraint $C5(c)$ considers constructor calls in constructors of confined classes. Since constructors are not inherited in Java, they have

to be in the same class or in the direct superclass (can be called via super(...)). This implies that *C5* depends only on the class itself and its superclass. When a class c is changed, *C5* is reevaluated for c and all direct subtypes.

- $C6(c)$ depends on all super-classes of c. Thus, it suffices to reevaluate $C6$ for all subclasses of c whenever c is changed. As an optimization, changes to c can be ignored that do not change c's supertypes.

- $C7(c)$ can change whenever the confined attribute of a type used in a public or protected field declaration of c changes. Since such a field type either was confined before the change or has become confined after the change, it has to be in the same package as c. Thus, whenever a type t changes *C7* needs to be reevaluated for all classes in the same package as t.

- The constraint $C8(c)$ checks return types of methods that are declared as public or protected. The strategy for evaluating $C8$ is the same as for $C7$.

For a given set of files that have been changed every constraint is processed separately. For every changed class the set of classes that have to be reanalyzed is computed and then the constraint is reevaluated against all classes in this set. This process is correct even if multiple changes have been performed, because it analyzes the same classes that would have been analyzed if an incremental analysis had been performed after every change.

By definition, the rules for computing the set of classes to be checked after a change guarantee that a constraint is reevaluated if any information it depends on has been invalidated. Hence, the value of all predicates is the same as if they had been evaluated by performing a whole-program analysis. Thus, our incremental algorithm is correct. Regarding its efficiency, with the current rules a constraint often has to be reevaluated for all subtypes of some type. Obviously, this may be a very big set. Suppose, for example, that the class Object is changed somehow. Now, constraints *A1–A3* for example have to be reevaluated for all subtypes of Object which essentially is every type.

A possible optimization is to use a call-graph analysis to reduce the reevaluations of constraints *A1* and *C4*. It is then possible to determine all method call statements that are affected by a given change. For the change from Listing 9.3, for example, the call-graph analysis would contain the information that the method called in line 17 has changed and it is possible to reevaluate

C4 for this location. This avoids having to check constraints *A1* and *C4* for all classes in a package. The challenge, of course, is to make the call-graph analysis incremental as the cost would be prohibitive otherwise and to make it fast enough to pay off compared to our current algorithm.

9.2.2.2 Prolog based

The Prolog based implementation of the type checking rules for confined type is a straight forward implementation of the constraints defined in Table 9.2 and in Table 9.1.

To achieve an acceptable performance the implementation relies on the automatic incrementalization feature of the underlying Prolog engine [SR06]. Hence, when compared with the Java based implementation, no explicit incrementalization of the analyses was necessary and it was immediately possible to start implementing the algorithms.

9.2.3 Evaluation

BAT is used as the base project for evaluating the performance of both implementations. The 790 classes of BAT are supplemented by 17 classes in three packages which implement a small part of a public key infrastructure. Initially, confined types were used in two of the three additional packages. Further, 12 anonymous methods were defined across the packages.

To assess and compare the incremental analysis times of both implementations 15 different source code changes were performed. The changes are described in Table 9.3 and resemble typical actions done by software engineers when developing and maintaining software. During the changes the classes in the third supplemented package were also made confined. The third column of Table 9.3 shows the number of violations that were introduced (+) or resolved (−) by a change; e.g., $+C8$ means that this change leads to one new violation of the $C8$ rule, $-3*C6$ means that three violations of the $C6$ rule are resolved.

No.	Description	Violations
1	Generated a public getter method for a confined field.	$+C8$

No.	Description	Violations
2	A small class is made confined.	$-A2$, $-A4$, $+C1$, $+C4$, $+C5$
3	Refactoring "Extract method.." in a medium-sized class. *(does not affect confined classes)*	
4	A small public class is declared confined.	$-C2$, $+C1$, $+C2$, $+C6$.
5	A new native method is created and declared anonymous.	$+A4$
6	A class with multiple subtypes is made confined.	$+C1$, $+12 * C2$, $+4 * C5$
7	A comment is updated.	
8	A call to a method that is not anonymous is made from within an anonymous method.	$+A1$
9	The type hierarchy is refactored (a class is no longer extended, instead an interface is implemented).	$-3 * C6$, $-C4$
10	Refactoring of a class's name. *(does not affect confined classes)*	
11	A new abstract class implementing two constructors of the extended class is created.	
12	The abstract class created in the previous step is made confined and used to replace a non-confined class.	$-12 * C2$
13	Deleted two no longer used classes.	$-C4$
14	A class's confined annotation is removed; instead four methods of the same class are declared anonymous.	$-C1$, $-C5$, $-C7$

No.	Description	Violations
15	A class high up in the hierarchy does no longer implement a small interface. *(does not affect confined classes)*	

Table 9.3: Code changes made to evaluate confined types

The performance measurements were made on an AMD Athlon XP 2600 notebook with 1024 MB RAM, Sun JDK 5 and SuSE Linux 9.3. Table 9.4 shows the result of the measurements. Since both analyses require the quadruples based representation of Java code the time to create the representation is shown in its own column. The time needs to be added to the analyses times to get the complete analysis time.[2]

The Prolog figures shown in Table 9.4 (column three) depict the raw performance figures of XSB to update its tables and to (re)run the queries to get the typing errors. The figures do not include the time required to create the Prolog encoding of the quadruples representation and — in particular — the time required by the Interprolog API [Int06b] to enable the communication between XSB and Magellan. Currently, the API is at its early stages and supports only a string based communication between the engine and client; i.e., to add a fact to the database the fact is encoded as a string, then the string is passed to the Prolog engine and decoded before being added to the Prolog database. The communication is further slowed down by using TCP/IP. The overhead caused by the current interface is approximately two seconds for our test cases.

Hence, using this interface it is currently not possible to run Prolog based analyses along with the incremental build process. However, the figures suggest that it is possible to use Prolog and to still get satisfactory performance. The analysis times are in general less than one second. The problems related to the communication are mainly engineering issues and can be solved by a tight integration of a Prolog engine with Magellan.

[2]To assess the performance gain provided by the automatic incrementalization feature the performance was also measured without tabling. Further, the time was measured with tabling, but without incremental maintenance of the tables. As can be expected, the evaluation times when using no tabling are practically constant and only depend on the number of packages with confined classes. However, even if only one small package contains a confined class the evaluation time is much too slow to run the analyses along with the incremental build process (larger than two seconds). Using tabling the evaluation times drop significantly, but are still between 3 and 10 times slower than the evaluation times of the incrementalized Prolog engine.

No.	3-address representation (msecs.)	Prolog (msecs.)	Java (msecs.)	Factor
1	7	176	3	59
2	8	195	5	39
3	36	171	2	86
4	6	168	6	28
5	8	194	2	97
6	12	555	124	4
7	36	210	3	70
8	14	348	4	87
9	11	291	4	73
10	211	663	81	8
11	121	144	1	144
12	11	297	84	4
13	3	216	7	31
14	16	230	3	77
15	26	552	3	184

Table 9.4: Confined types analysis times

A comparison of the analyses times of the Prolog and the Java based implementation shows that the performance of the Java based implementation is by far superior to the Prolog variant. However, implementing the analyses in Prolog took approximately two days work, while the Java implementation took roughly 3 weeks. Further, since the analyses are automatically incrementalized when using Prolog, it was not necessary to develop an incremental algorithm and reason about its correctness.

9.2.4 Related Work

When dealing with aliasing, four categories of work are considered [HLW+92]: detection, prevention, control and advertisement of aliasing. Confined types mostly fall under the category of prevention and control.

The notion of alias protection for object-oriented languages was introduced by Hogg [Hog91] in order to enable modular reasoning for groups of classes. These groups are called *islands* and ensure the restriction of alias-

ing to classes on the island. Hogg differentiates between static and dynamic aliases. Static aliases are aliases via instance variables and dynamic aliases are those via parameters or local variables. Static aliasing can lead to undesired side effects in later invocations of the aliased object. Dynamic aliases were seen as unproblematic, because they disappear at the end of the execution of the method in which they are defined. Means to control static aliasing were introduced with islands. Islands are the transitive closure of a set of objects accessible from a *bridge* object. A bridge object is the sole access point to a set of instances that make up an island.

To ensure that no static aliases are created from outside the island to objects on the island, the methods of the bridge object are restricted. Only methods with parameters and return values that either do not modify the state of the system, or have only parameters and return values that have at most one static alias are allowed. This avoids the creation of unwanted aliases. For example, a return value of a method can be tagged with *unique* to state that exactly one reference to its value exists. The value can only be assigned to other variables, if the original reference is released.

The full encapsulation of aliases of this approach is too restrictive for many common design idioms used in OO programming. For example, no object could be a member of two collections simultaneously if either collection was fully protected against aliases. In this case, one collection would be an island, prohibiting that references to its members show up outside the island.

In [NVP98] Noble et al. present a more flexible approach to control aliasing when compared with *islands*. The taken approach is to enable aliasing by introducing explicit aliasing modes. The authors differentiate between the *representation* of an object, which corresponds to its fields, and *arguments*, which are parameters to methods of the object. The representation of objects should only be accessible via the object's interface. In Java, for example, fields would have to be marked as `private` and aliases to them should not be returned via getter methods. The state of the object should only depend on arguments with an immutable state. If the state of the object was dependent on the mutable part of arguments to its methods, the state of the object could be changed by changing the state of the arguments long after the call, bypassing the object's interface. The approach uses tags to annotate types and enables the compiler to enforce the restrictions mentioned on the creation of aliases. A formalization of this model is discussed by Clarke et al. [CPN98]. Even though both approaches enable flexible alias control, they are designed for a language without inheritance or subtyping.

A variant of ownership types is used by Boyapati et al. [BLR02] to prevent

data races and deadlocks by partitioning locks into a fixed number of equivalence classes and specifying a partial order among these equivalence classes. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in descending order. Ownership types are used to ensure that the locks that protect an object also protect its encapsulated objects.

The approach of Clarke et al. [CRN03] implements a confinement checker for Java to solve the domain specific problem of passing a this reference from one Enterprise Java Bean component to another component, as discussed in the introduction. While confined types are a generic solution to control aliasing, Clarke et al.'s approach solves an EJB specific problem.

The work of Fong [Fon04] describes how to translate the notion of confinement, which is formulated for static analysis of Java source code, to dynamic analysis of Java Bytecode. The approach retains the confinement annotations made in the source code at bytecode level. This enables link time checks of confinement rules. It also describes a form of secure cooperation between mutually suspicious code units, where, for example, a resource object can be shared between two untrusting modules while ensuring its confinement to a given domain. The implementation extends the runtime of the Pluggable Verification Modules of the Aegis Research JVM. Our approach uses static analysis to ensure the confinement properties at compile time and to immediately inform the user of confinement violations.

In [ZPV03], the notion of confined types is formalized in the context of Featherweight Java (FJ). In FJ, confined types are extended to confined instantiations of generic classes.

Reverse engineering approaches to the detection of aliasing are described in [GPV01, PNB04]. Kacheck/J [GPV01] is a tool to infer confinement in Java code and was used to test the thesis that all package-scoped classes in Java programs should be confined. About 25% of the package scoped classes of their benchmark suite were confined anyway and 45% could be refactored to be confined just by changing visibility modifiers. These numbers are supported by the findings of Potanin et al. [PNB04]. They presented metrics of uniqueness, ownership and confinement by analysing snapshots of Java program's object graphs and found that a third of all objects were strongly confined.

9.3 Conclusions

This chapter discussed using MAGELLAN for implementing optional type systems. The first part discussed the integration of advanced type systems with existing languages. Following the discussion, a Java as well as a Prolog based implementation of confined types as a pluggable type system was presented. Both approaches were then thoroughly evaluated.

Based on the evaluation, we can draw the conclusion that MAGELLAN is well suited for the implementation of new type systems. In particular, MAGELLAN supports: (a) rapid prototyping new type systems and (b) implementations with strict performance requirements.

A detailed discussion of the advantages is given in the following:

- The Prolog integration provided by MAGELLAN is particularly well suited for testing new language concepts. As shown by the evaluation, the Prolog interface and the SSA representation of the code facilitates the rapid prototyping of type checkers. Enabling researchers to focus on the correct definition of the typing rules and by freeing them from implementation issues, which are prevalent when using Java, many tedious and error prone activities are avoided. When using Prolog, e.g., it is not necessary to reason about the correctness of the incrementalization as this is done automatically. Further, the performance is sufficiently fast to analyze mid-sized projects in reasonable time. Hence, using Prolog it is also possible to make preliminary assessments of type checking algorithms w.r.t. to their scalability and performance characteristics.

- Using Java it is possible to build high-performance implementations of (pluggable) type systems on top of MAGELLAN. Even analyses that require precise intra-procedural data- and control-flow information take no more than 150 milliseconds for analyzing $\approx$ 900 methods and constructors. Hence, it is possible to always run these analyses along with the incremental build process.

- MAGELLAN is already delivered with sophisticated static analyses, which are often required to implement type checkers; in particular intra-procedural data- and control-flow analyses are readily available.

- Using MAGELLAN the user can activate multiple optional type systems simultaneously as the overhead caused by each additional type system is small. For example, the type system proposed by Fähndrich and

Leino [FL03] for non-null types is orthogonal to confined types and it is reasonable to assume that both (optional) type systems are used in the same project. Since, both type systems require the same intra-procedural data flow information they will probably use the same code representation, e.g., BAT's SSA form based representation. As the estimated effort for type checking non-null types is similar to the effort necessary for checking confined types, the additional amount of time required when both type systems are active will be small. The expensive analyses to create the SSA form are executed only once and the results are used by both analyses.

Being able to run multiple optional type systems in parallel can also be used during the design and implementation of new type systems. For example, to test if a Prolog based implementation and a Java based implementation of the same type system both report the same errors; i.e., both analyses are run in parallel and the results are compared.

- By implementing advanced type systems on top of MAGELLAN it is possible to avoid bloated compilers and to ensure that the type systems required by the applications can be introduced when needed. E.g., a type system to enable safe concurrent programming [BLR02] or to prevent data races [BR01] is only required for multi-threaded applications.

  Moreover, in most cases several different variants to solve a similar problem exist, e.g., to control aliasing. In this case, the most well suited variant can be chosen and it might even be possible to use multiple type systems for a similar purpose in different parts of the same application. For example, using flexible alias protection [NVP98] for implementing container classes (e.g. Set, List, or Map) and using confined types [VB01] for implementing, e.g., a component container.

  Hence, MAGELLAN provides a foundation on top of which modular languages [Bra04] which enable a customization of a core language to a project's needs can be built.

- By building upon MAGELLAN it is possible to focus on the implementation of the programming language concept. It is not necessary to take care of the integration with the incremental build process, to execute the required base analyses, to enable a navigation between error messages and the corresponding source code, or to take care of the visualization of the error messages.

Part IV

Summary

Chapter 10

Conclusions

> Don't fear failure so much that you refuse to try new things. The saddest summary of a life contains three descriptions: could have, might have, and should have.
>
> *Louis E. Boone*

The goal of this thesis was to develop concepts and techniques for open integrated software development and analysis environments to improve (a) the productivity of developers and (b) the quality of software. After an investigation of the current landscape of software comprehension and analysis tools, it became evident that a large number of successful tools already exists and that the most immanent problem was *lack of integration*.

Specifically, inter-tool integration, integration into IDEs and, most important, integration with the incremental build process of modern IDEs was broadly missing. Hence, the potential of a tight tool integration with an IDE and its build process — *providing developers with timely information* — was not leveraged.

Especially, a fundamental approach to the integration of several tools with an incremental build process was missing; a direct integration of several independent tools with the incremental build process is not feasible since the memory requirements and the overall analysis time required by the tools would be prohibitive.

To enable the simultaneous integration of several software engineering tools into an incremental build process this thesis developed the concept of

an *open static analysis platform integrated into the build process*. Such a platform coordinates the execution of an open set of analyses such that a source model is specifically derived for the needs of tools executed on top of the platform. The novel concept is that tools share the same source model and that the data model of the software to analyze is not fixed, but rather derived from the needs specified by individual analyses to run. An open data model facilitates the addition of new base analyses and, hence, is indispensable for an open platform. By sharing the source model the overall analysis time and the amount of required memory can be reduced.

As demonstrated by the prototypical platform MAGELLAN and the tools built on top of it, the proposed concept is feasible and does facilitate the simultaneous integration of several analyses into the incremental build process. Hence, the goal of this thesis is achieved: Compared to using plain IDEs it is possible to provide developers with a much wider range of timely information. The vision of *Open Integrated Development and Analysis Environments* is prototypically realized.

Based on the experience gained while implementing the tools for evaluating the platform, the following additional conclusions can be drawn:

- MAGELLAN and the tools built on top of it validate the study that lead to the identification of the requirements on open static analysis platforms (Chapter 2).

  The study analyzed software comprehension and analysis tools to determine the requirements on platforms that should serve as a foundation for both categories of tools. It is certain that all important requirement were identified since it was demonstrated that a wide range of comprehension and analysis tools can be developed on top of a platform (MAGELLAN) that fulfills a subset of the identified requirements and does not provide additional functionality.

- The LSV and the ASL language (Chapter 3) enables reasoning about the source model at an abstract level. This makes it possible to use the platform even if some data is actually stored in an external database, e.g., in a Prolog database.

- The high-level automatic parallelization of analyses that process disjunct data sets, already leads to significant performance gains.

- MAGELLAN demonstrates that it is possible to develop a platform that is flexible enough to support a wide range of different tools without

sacrificing performance. This is an indispensable prerequisite to enable a tight integration with the incremental build process.

It was demonstrated that executing more than 40 analyses — including analyses which perform intra-procedural data- and control-flow analyses — does not lead to a prolongation of the build process that can be perceived by developers in day-to-day work.

- Query engines that do not evaluate queries incrementally do not provide the performance necessary to enable the regular execution of queries as part of the build process.

  The performance evaluation of the XQuery engine has shown that it takes too long to evaluate non-incremental XQuery queries regularly as part of the incremental build process. However, the automatic incrementalization of Prolog based queries as performed by the embedded Prolog system is effective and enables the evaluation of Prolog queries along with the incremental build process.

- Even if the performance of a query engine is not sufficient to enable build process integration, the performance requirements of on-demand analyses are less strict and, hence, embedding such engines is useful as it provides developers with more opportunities.

- Query engines that support query chaining are the first choice for the implementation of software comprehension tools where results of queries are reused as input for the execution of subsequent queries.

- The prototypes built on top of Magellan: QScope [EGM+06], Sextant [EHMS05], the annotation based Checkers [ESM05], and the Confined Types type system [EKMS06] demonstrate that open static analysis platforms facilitate research. Platforms such as Magellan are an ideal testbed for the development of all kinds of new analyses, new type systems or completely new categories of tools, e.g., for analyzing the interaction between aspects [FT06, SSF06]. Such platforms can also serve as a foundation for an incremental model checker, as outlined in [CNDE05].[1]

[1] Model checking a program is typically done in a multi-step process where the first steps are usually transformation of the source code in some form of higher level intermediate representation.

- By building tools upon analysis environments it is possible to focus on the tool's distinguishing functionality; base functionality, such as, the integration with the build process, source code parsers and querying support, can be reused. Hence, such platforms improve the productivity of developers and researchers building software engineering tools. The possibility to reuse base components is also an incentive for engineers to develop towards such a platform. It is conceivable that such a platform can lead to a community where some researchers and developers contribute (base-)analyses and others (re)use them to build more advanced software engineering tools.

- Analysis environments enable a strict separation between compilers and analyses. It is no longer necessary to build analyses into compilers to provide developers with more information about possible issues in the source code. Hence, bloated compilers can be avoided and analyses required by the applications can be introduced when needed.

- Query engines facilitate prototyping new analyses. Compared to implementations using imperative languages, analyses implemented as queries are more concise and can be developed and tested in a shorter time. Prototypical implementations of analyses as queries make a first assessment of analyses w.r.t. to their usefulness and scalability possible.

  Furthermore, analysis environments can make the development of an optimized (imperative) implementation of an analysis easier if a query based implementation of the same analysis is available. In this case, it is directly possible to compare the generated error warnings and messages of both approaches, and to test and compare the effect of (incremental) changes on the analyses. Testing if the result of two implementations is identical can easily be implemented and then the first implementation can be used as a testbed for the second implementation.

- The integration of analysis and development tools into one environment enables users to directly use the gained knowledge to maintain and evolve their code. The technological gap between these tools is removed.

Chapter 11

Future Work

> The best way to predict the future is to invent it.
>
> *Alan Kay*

As demonstrated in this thesis, the logical structure view (LSV) and the analysis specification language (ASL) provide sufficient expressiveness for specifying the effect of many analyses. Nevertheless, some use cases are not well supported at the moment. For example, assuming that there are two analyses that both specify to transform annotations, i.e., both specify to invalidate an entity Annons. In this case, the scheduler will schedule at most one of these two analyses, as they have conflicting requirements. But, if the analyses actually transform two different annotations $@X \rightarrow @X'$ and $@Y \rightarrow @Y'$ then the analyses are not conflicting. Supporting analyses that depend on the same type of information (e.g., annotations), but always operate on different (object) instances (e.g., disjunct sets of annotations), is left for future work.

As also shown in this thesis, the automatic parallelization of analyses is effective and leads to an improvement of 28% in average. Nevertheless, further improvements can be achieved by dropping the requirement that each analysis is strictly executed in one particular time slot. For example, given three analyses A, B, and C, where A is a long running analysis without any dependencies, B is a short running analysis also without any dependencies and C is an analysis that depends on B. In this situation the scheduler would determine that A and B can be executed in parallel and would assign the

same slot to A and B. The analysis C would be scheduled for a later execution slot. At runtime — assuming that we have a multi-core / multi-CPU system — the analyses A and B would be executed in parallel. But, when analysis B finishes the dispatcher will not immediately start with analysis C, even though all specified dependencies are fulfilled. Instead, the dispatcher will wait on the completion of analysis A before C is started. Hence, an area of future work is to improve the flexibility of the scheduler to increase the parallelization.

Another area of future work concerns resource usage. The amount of memory required to represent the source model of a software system is limiting the size of programs that can be analyzed. Hence, research on compact program representations is important to enable the analysis of ever growing programs. In this context, research on compact representations of XML information is of particular importance. XML is already widely used by software engineering tools and, more important, many developers are familiar with XML technologies. Hence, supporting XML is important for tool adoption. Some preliminary results w.r.t. limiting the amount of required memory when using XML are outlined in Appendix V, but further research is necessary to achieve scalability to very large programs.

The embedded Prolog system has shown that automatic incrementalization of queries is effective and that significant performance gains can be achieved. The achieved performance is even sufficient to always execute such queries along with the incremental build process. However, the performance is far from being up to par with manually incrementalized analyses and, hence, only a much smaller number of queries can be executed as part of the incremental build process. Thus, an area of future work is to further improve the automatic incrementalization of queries. One possibility could be to investigate if it is possible to decide — by statically analyzing queries — if a change to the fact base might have any effect on any query or not. For example, if a developer adds the volatile modifier to a Java field the corresponding fact changes. But, if no query analyzes the corresponding argument of field facts, it is useless to reevaluate any query — including those that analyze fields.

Part V

Appendix

BAT Based Checkers

The checkers presented in this chapter were implemented as part of the evaluation of MAGELLAN.[1]

Class Interface Related Checkers

The following checkers use the code representation generated by BAT [BAT06].

Covariant compareTo() method defined
: Searches for classes that implement the interface Comparable and that define a compareTo method with a formal parameter different from java.lang.Object.

  This may lead to unexpected results when an instance of the class is put into a sorted collection (e.g., a SortedSet). In this case, the entries are still sorted using the compareTo(Object) method; the newly defined method does not override the superclass's compareTo method.

Covariant equals() method defined
: Searches for classes that define an equals method with a formal parameter different from java.lang.Object.

  If an instance of such a class is put into a set (e.g., a HashSet) the result might not be as expected, as the method does not override the original equals method defined by java.lang.Object and, hence, is not used by the collections API.

Field shadows field in superclass
: Searches for fields that shadow field declarations in superclasses. Such field definitions hinder software evolution and maintenance; even when both fields are private it might confuse developers.

[1] The author would like to thank Benjamin Rank for implementing the checkers as part of his Diploma Thesis.

Violation of Object's equals()-hashCode() contract

Checks that the methods boolean equals(Object) and int hashCode() are implemented pairwise.

If only one of these methods is implemented the corresponding contract defined by java.lang.Object is violated. It might happen that two objects have different hash codes even though both equals methods return true. In this case, data structures, such as a HashMap, which rely on the contract will have an unpredictable behavior.

Non private field has getter or setter

Searches for non-final fields that are not private and that can also be accessed by getter or setter methods. This leads to ambiguities how to access the field.

Method Implementation Related Checkers

The following checkers use the 3-address based code representation in static single assignment form generated by BAT [BAT06].

Violation of call restriction

It is checked that methods annotated with the @Restrict annotation are only called by classes with fully qualified names matching the regular expression specified by the annotation.[2]

For example, the method shown in Listing 11.1 may only be called from within classes matching the regular expression. This means that only classes within the package de.tud.bat.io. or any subpackage are allowed to call the method.

```
1 @Restrict(value = "de\.tud\.bat\.io\.\\.*")
2 public void restricted() { ... }
```

Listing 11.1: Usage of the @Restrict annotation.

Sometimes it is necessary to make a method public or protected visible, though the method is not meant to be part of the public interface of the class. In these cases the @Restrict annotation can be used to communicate and enforce the intended design.

[2]The checker also checks that only non-private methods are annotated using @Restrict as it is useless to annotate private methods.

Comparison of two strings by reference

Searches for String comparisons using "==" or "!=". This checker is similar to the generic reference comparison checker. The only difference is its focus on String objects and that String constants are also considered.

Comparison of two different types using Object.equals()

Searches for calls to equals where the object passed as the argument has a different type than the receiver of the call. This is a bug pattern in most cases as the result is most probably always false.

Explicit invocation of Object.finalize()

Searches for explicit calls to the finalize() method. It is a best practice not to explicitly call this method, as it is supposed to be called only by the garbage collector.

Exceptions must be made explicit

Checks that a method explicitly declares all exceptions that it might throw — in particular RuntimeExceptions.

By design the compiler does not enforce that RuntimeExceptions are handled or declared. But, to support comprehension of a method listing all potential exceptions is advantageous.

Field should be accessed using its getter or setter

Searches for (private) fields that are directly accessed despite being accessible using getter or setter methods. Consistently accessing a field fosters software evolution and maintenance.

To avoid too many false positives it is checked that the setter and getter methods are "trivial"; i.e., the getter just returns the field's value and the setter just updates the field's value. In case of non-trivial getter/setter methods directly accessing the field is usually a deliberate choice and no warning is not reported.

The finalize() method does not call super.finalize()

Searches for finalize() methods that do not call super.finalize() for every possible control flow path. In such a case the finalization of the object might be incomplete.

Invocation of hasNext() inside next()

Searches for classes which invoke a next() method from within a boolean

hasNext() method. This violates a best practice, as the developer usually expects that calling hasNext() does not change the state of the iterator.

If is constant

Searches for if statements where the result of the comparison is statically known to be constant (either true or false). Such cases often indicate bugs or a lack of understanding Java. An example of a hard to detect instance of this bug pattern is presented in Listing 11.2.

```
1 Object o = null;
2 try {
3   o = System.in.read();
4 } catch (Exception e) {
5   if (o != null)
6     System.out.println(o);
7 }
```

Listing 11.2: Example of an if statement where the expression is constant

The expression o != null in Line 5 will always be false, because o is null in case that an exception is thrown by System.in.read(). If System.in.read() does not throw an exception the assignment of the method call's return value to o (Line 3) will never throw an exception. Hence, if the catch block is reached o is null.

InputStream must be closed

Checks that a newly opened InputStream (or one of its subclasses) is closed on all subsequent control-flow paths.

This checker only analyzes the intra-procedural control flow graph, i.e. if the stream is passed to another method and closed in that method this checker will report a false error. However, it is also a best practice to open and close streams in the same method. False errors can be considered as hints for further refactorings.

Result of integer division casted to double

Searches for integer divisions where the resulting value is immediately casted to a double value. In most cases the developer intended to perform a double division. For example, if a is 1 and b is 2 then the value of d, given the statement double d = a / b, is 0.0d. In case of the statement double d = (double) a / b the value is 0.5d.

Result of Math.random() casted to int

Searches for calls to Math.random() where the return value is immediately casted to int. In this case, the value will be zero as the function always returns a value between 0.0d and 1.0d.

Never invoke Object.wait(...) outside a loop

Checks that the Object.wait(...) methods are always invoked inside a loop. This idiom is described in detail in Effective Java [Blo01] and is also briefly described in the Javadoc comments of the wait methods.

One character appended as String

Searches for strings that have only one character and that are concatenated with other strings. It is more efficient to concatenate a single character instead. E.g. the code generated and executed in case of Line 2 of Listing 11.3 is more efficient than the code shown in Line 1.

```
1 String result = "Hello" + name + "!";
2 String result = "Hello" + name + '!';
```

Listing 11.3: Appending one character to a String

Redundant call to String.toString()

Searches for calls of toString() on instances of java.lang.String; these calls are superfluous.

Reference comparison

Searches for comparison of objects using == or !=. In these cases reference identity is used for the comparison which is often a cause of bugs, as exemplified in the following listing.

```
1 Integer i1 = new Integer(1);
2 Integer i2 = new Integer(1);
3 if (i1 == i2) {
4   System.out.println("i1 and i2 represent the same value"); // not reached
5 }
```

Return value ignored

Searches for calls to methods where the method's return value is ignored. In some cases ignoring the return value makes the whole method call useless. For example, the method String.concat(String) does not change the String object that is the receiver of the call. Instead, a new

String object is created which stored the concatenated String. Hence, if the return value is ignored the method invocation has no effect.

Return value must not be ignored

Finds calls to methods where the return value is ignored and where the called method is annotated using the @ReturnValueMustNotBeIgnored annotation. This checker is similar to the "Return Value ignored" checker, but does not produce false warnings as only explicitly annotated methods are checked.

For example, if the return value of the method in Listing 11.4 is ignored, an error message is generated.

```
1 @ReturnValueMustNotBeIgnored("The concatenated value is returned; the
      state of this class is not changed.")
2 public int concat(String s) {
3   return this.toString()+s;
4 }
```

Listing 11.4: A method where the return value must not be ignored

String concatenation in StringBuilder.append()

Searches for Strings that are concatenated using "+" and which are then passed to StringBuilder.append(String), e.g., <StringBuilder>.append("abc"+"def") This is highly inefficient, using the existing StringBuilder would be more efficient.

String.substring(0) returns whole string

Finds calls to the String.substring(int) method where the parameter value is 0. In this case the whole String is returned so that the invocation is not necessary. This checker performs an intra-procedural analysis only.

Comparison of floating point values using == or !=

Finds comparisons of floating point values using == or !=. These types of comparisons are known to be error prone; due to the limited precision of floating point values often false is returned even if true would be mathematically correct. For example, given float m = 0.01f, n = 0.1f; m *= 10.0f; the result of the comparison m == n is false; which is not expected by most programmers.

Unnecessary type check using instanceof

This is an intra-procedural analysis that searches for unnecessary type comparisons using the instanceof operator. For example, in Listing 11.5 it can statically be determined that *o* is of type Integer in Line 3. This analysis performs an intra-procedural analysis only.

```
1 Object o = new Integer(1);
2 ... // instructions not changing ''o''
3 if (o instanceof Integer) {
4   System.out.println("Integer");
5 }
```

Listing 11.5: Unnecessary instanceof operator

Uninitialized private field

Searches for private fields that are not explicitly initialized within their declaring classes. This case most often indicates a bug.

Passing a String object to String's constructor

Searches for the creation of new String objects that are initialized with a String object. Such calls are useless because String objects are immutable and, hence, it is sufficient to continue using the "old" String.

Useless control-flow in method

Searches for control flow statements branching to the same instruction in all cases. For example, the misplaced semicolon after the condition in Listing 11.6 makes the if statement useless.

```
1 if (a);
2   ...;
```

Listing 11.6: Useless control-flow statement.

BAT_2XML: an XML Representation of Java Bytecode

Part of the material in this chapter is published in: BAT_2XML: XML-based Java Bytecode Representation [Eic05].

BAT_2XML is a library to create XML representations of Java class files. The library is used by several of the tools developed on top of MAGELLAN. BAT_2XML enables the creation, transformation and querying of Java bytecode [LY99].[3] The transformation and creation related features of BAT_2XML are, however, not presented in this section as they are not relevant for developing analyses on top of the representation.

The XML representation is close to a one-to-one representation of the corresponding Java bytecode and is readily comprehensible by developers familiar with bytecode. For example, the code shown in Listing 11.7, which simply prints "HelloWorld" to System.out, has the XML representation shown in Listing 11.8.

```
1 public class HelloWorld extends java.lang.Object
2
3 public static void main(java.lang.String[]);
4  0: getstatic // java/lang/System.out:Ljava/io/PrintStream;
5  3: ldc // String HelloWorld
6  5: invokevirtual // java/io/PrintStream.println:(Ljava/lang/String;)V
7  8: return
8 }
```

[3] Besides being used in the context of MAGELLAN, $BAT_2XML$was also used in the context of aspect-oriented programming [KLM+97] to implement, so-called, aspect weavers [EMO04, EM05].

Listing 11.7: Java bytecode of “HelloWorld”

The XML representation abstracts from some details of Java bytecode, for example:

- bytecode offsets (shown in Listing 11.7, Lines 4–7) are omitted.
- bytecode instructions that operate on multiple types are replaced by sets of instructions where each instruction is specialized for one specific type. For example, the generic ldc (load constant) instruction shown in Listing 11.7, Line 5 is replaced by a stringconst instruction as shown in Listing 11.8, Line 8.
- the type information is represented in fully qualified form as used in Java source code (e.g. as shown in Listing 11.8, Lines 4,7,11).

Besides from these minor differences each bytecode instruction is represented by a corresponding XML element.

```
1 <class name="HelloWorld" sourcefile="HelloWorld.java" visibility="public">
2   <inherits><class name="java.lang.Object" /></inherits>
3   <method name="main" visibility="public" static="true" >
4     <signature><parameter type="java.lang.String[]" /></signature>
5     <code>
6       <get declaringClassName="java.lang.System" fieldName="out"
7            staticField="true" type="java.io.PrintStream"/>
8       <stringconst><value>HelloWorld</value></stringconst>
9       <invoke declaringClassName="java.io.PrintStream"
10                methodName="println" >
11          <signature><parameter type="java.lang.String" /></signature>
12      </invoke>
13      <return />
14    </code>
15  </method>
16 </class>
```

Listing 11.8: XML representation of “HelloWorld”.

As illustrated in the example, the XML representation abstracts from some details of Java bytecode. The chosen abstractions support comprehension of the representation by developers not familiar with Java bytecode

and ease the development of analyses. The differences are described in the following.

- In BAT$_2$XML all information is resolved, that is, the Java bytecode constant pool is completely hidden. Further, all types are represented using the same form as used in Java source code, e.g.,java.lang.Object and not java/lang/Object as used in Java class files.

- In BAT$_2$XML the overall number of instructions is reduced.

  For example, the Java bytecode defines three different instructions to create new arrays: newarray, anewarray, multianewarray, and, to make the situation even worse, the multianewarray instruction can also be used to create one-dimensional arrays; the (a)newarray instruction just exists for optimized runtime performance.

  Another example are the different instructions that can be used to push the int value "0" onto the stack. The Java bytecode provides the specialized instruction iconst_0 and the generic iload instruction where the value is explicitly specified. In BAT$_2$XML both cases are represented using intconst with the value as a parameter.

  In general, BAT$_2$XML abstracts from the differences between closely related instructions by providing only a representation of the most generic instruction. Specialized instructions are always represented as parametrizations of generic instructions.

- In BAT$_2$XML each instruction serves exactly one purpose. For example, instead of having one generic ldc instruction to put different types of constant values onto the stack, a specialized instruction is defined for each type. E.g., for string values a stringconst instruction is defined.

- The distinction made in the Java bytecode between values that occupy one (e.g., int, short, or address value) or two stack items (double, long) is not made in BAT$_2$XML.

  This concerns the dup and pop bytecode instructions which duplicate or pop a specific number of stack items; a stack item is always 4 Bytes while a value has either 4 or 8 bytes. In BAT$_2$XML these instructions are replaced by dup and pop instructions that directly specify the number of values that are duplicated or popped. For example, the Java bytecode instruction dup2 duplicates only one value if the type of the top most value on the stack is long or double and duplicates two values

in all other cases. In BAT$_2$XML the first case is represented by a dup instruction where the value of the attribute that specifies the number of duplicated values is 1, respectively 2 in the second case.

- The target of the jump instructions goto, switch and if is specified by referring to the id of the target instruction and not by using bytecode addresses and relative offsets. E.g., an if-else structure is represented as shown in Listing 11.9. The target of the if instruction is either the get instruction, which has the id m2i0, or the instruction immediately following the if instruction, if the condition is not satisfied. The target of the goto instruction is the return instruction. In short, an id attribute is used to mark a jump target and an idref attribute is used to reference it.

```
1 <if operator="ne" idref="m2i0" />
2 ...
3 <goto idref="m2i1" />
4 <get ... id="m2i0" />
5 ...
6 <return id="m2i1" />
```

Listing 11.9: XML representation of jump instructions

- BAT$_2$XML performs a control flow analysis to make an analysis of (intra-method) subroutines easier. BAT$_2$XML determines the jump target of a subroutine's ret instruction in relation to the jump to subroutine (jsr) instruction. For example, in Listing 11.10 the ret instruction (Line 9) lists all jump targets in relation to the jsr instruction which called the subroutine (Line 10, 11). If the subroutine was called by the jsr instruction with the id JSR1 (Line 1) the target of the ret instruction (Line 9) is the instruction with the id i0 (Line 2).

```
1 <jsr id="JSR1" idref="i3" />
2 <invoke ... id="i0" />
3 ...
4 <jsr id="JSR2" idref="i3" />
5 <get ... id="i2" />
6 ...
7 <store ... id="i3" />
8 ...
9 <ret index="..." />
```

```
10   <path><caller idref="JSR1" /> <target idref="i0" /></path>
11   <path><caller idref="JSR2" /> <target idref="i2" /></path>
12 </ret>
```

Listing 11.10: XML representation of Java bytecode subroutines

- Additionally to the information defined in Java class files, the control flow graph of a method is explicitly represented to make code analysis easier.[4]

  The control flow graph for the method abs shown in Listing 11.11 is depicted in Figure 11.1. The XML representation of the method is given in Listing 11.12.

```
1 public int abs(int value){
2   if (value < 0)
3     return −value;
4   else
5     return value;
6 }
```

Listing 11.11: Definition of an abs function

```
1 <load index="1" fg:bb_idref="m2bb0" />
2 <if operator="ge" fg:bb_idref="m2bb0" idref="m2i0" />
3 <load index="1" fg:bb_idref="m2bb1" />
4 <neg fg:bb_idref="m2bb1" />
5 <return fg:bb_idref="m2bb1" />
6 <load index="1" id="m2i0" fg:bb_idref="m2bb2" />
7 <return fg:bb_idref="m2bb2" />
8
9 <fg:flow_graph>
10   <fg:bb fg:id="m2bb2"> <fg:pre fg:idref="m2bb0" /> </fg:bb>
11   <fg:bb fg:id="m2bb0"> <fg:succ fg:idref="m2bb2" />
12                          <fg:succ fg:idref="m2bb1" /> </fg:bb>
13   <fg:bb fg:id="m2bb1"> <fg:pre fg:idref="m2bb0" /> </fg:bb>
14 </fg:flow_graph>
```

Listing 11.12: XML representation of abs

[4]The control flow graph visualizations were generated with a small stylesheet which transforms the XML representation of the graph in a *DOT*[GKN02] file for generating the visualizations.

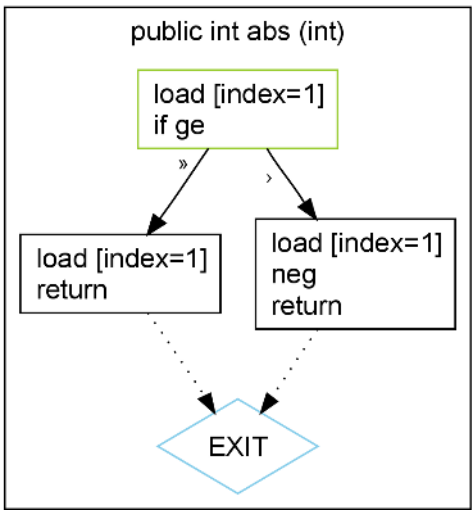

Figure 11.1: Control-flow graph of abs

The control flow graph is encoded by explicitly specifying the predecessors (Line 10 and 13) and successors (Line 11 and 12) for each block (Listing 11.12, Lines 9–14). Further, every instruction (Listing 11.12, Lines 1–7) is associated with the id (bb-idref) of its block.

Coping with XML Related Scalability Issues

One of the major issues when using XML is the memory required for keeping XML data in memory. Initially, using a standard Java API (such as JDOM [HM06]) the memory requirements for keeping the complete XML representations of a few hundred class in memory was prohibitive. Additionally, many analyses (queries) also require information about classes in the Java runtime library which made the situation even worse.

To tackle the "memory requirements problem", the following optimizations were carried out:

1. Details about the method implementations of library classes were omitted. Though, analyses which would require a complete representation of library classes are no longer supported, this is not considered a severe restriction: The XML representation is primarily used by software comprehension tools that usually do not analyze the implementation of library classes.

2. An optimized version of the JDOM [HM06] library, which is used for keeping the XML data in memory, was developed. The optimized library is called JDOM$_{opt}$ in the following.[5] Compared with the original library two important changes were made:

   - The names of all XML elements are *internalized*, i.e., the String objects which represent the names of the XML elements and attributes are cached and a reference to a String object in the cache is used if the String is already in the cache, otherwise the new String object is put into the cache for later usage.

[5] JDOM$_{opt}$ is 100% interface compatible with JDOM. Hence, to make use of the optimizations it is sufficient to exchange the libraries.

- The structural properties of BAT$_2$XML generated XML documents were used to control the initial size of each element's lists (java.util.ArrayList) that keep references to child elements and attributes. The XML elements of documents created by BAT$_2$XML have ≈4 attributes and the generated tree is flat. Furthermore, few elements have large numbers of children and most elements are leaf nodes; basically all elements that represent bytecode instructions are leaf nodes. Hence, it is evident that the initial size of "10" for the lists which store an element's attributes and child elements, is either too large or much too small. Extensive experiments showed that setting the initial size to "0" provides the best trade-off between reducing the memory requirements and the loss in performance due to the more frequent resizing of the lists.

Taken together these changes led to a ≈ 50% reduction of the required memory. As shown by the bars in Figure 11.2, using JDOM requires approximately twice as much memory as JDOM$_{opt}$. For comparison, the memory requirements when using the XML library XOM [Har06] as well as using BAT's own internal representation are also shown. BAT$_{opt}$ is a variant of BAT where all information is stored using arrays instead of ArrayLists.

As shown by the graph in Figure 11.2, when using JDOM$_{opt}$ the time to read in the classes is increased by ≈ 40% compared to JDOM. In the context of MAGELLAN, the increase in the processing time is less important, because a typical incremental build only affects a very small number of classes: most often just one or two classes.

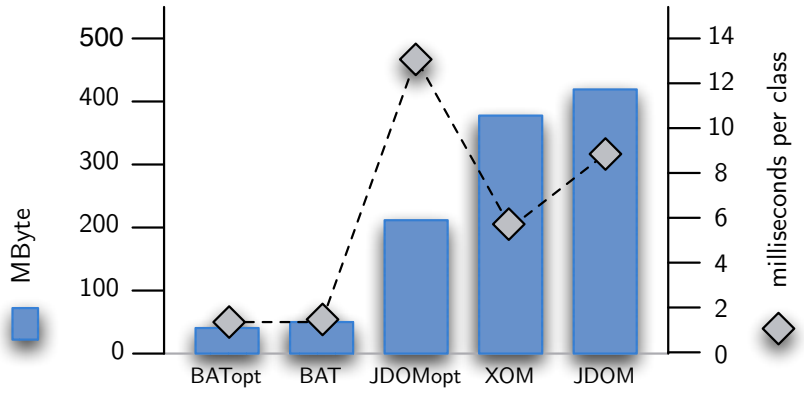

Figure 11.2: Memory requirements when using XML representations

Even though the memory requirements using JDOM_{opt} are much better than those of JDOM and enable to analyze projects with several hundred classes, analyzing projects with more than $\approx$ 1000 classes is hardly possible.

To further reduce the memory requirements of the XML database and to reduce the processing time, preliminary experiments using a thin wrapper layer were made. For each class of BAT_{opt} a wrapper class was implemented that implements the org.w3c.dom.Element interface. The wrapper object uses the underlying BAT object as its data store and hardcodes the type of the element. For example, for the BAT class Method a wrapper class MethodW was implemented that always represents <method> elements. In case that a query requires the name of a method, the wrapper then lazily creates a specialized XML attribute that represents the name of the method.

Due to the implementation of the standard W3C interface it is possible to use standard conform XML libraries for transformation[6] and querying.

Even though the implementation is not complete — no wrapper classes for bytecode instructions were developed — the preliminary results are encouraging. A first estimation shows that keeping a BAT_{opt} object and its wrappers in memory requires at most 60% of the memory needed by JDOM_{opt}. These 60% also represent the worst case, because the wrapper objects are created lazily and many elements and attributes are never queried. The additional time required to create the wrapper objects is $\approx$1,72 milliseconds per class, i.e., creating the BAT_{opt} representation and all wrappers is $\approx$45% faster when compared with the fastest XML solution (using XOM).

The implementation of the wrappers was, however, not completed as the implementation is extremely cumbersome. The W3C's Element interface defines more than 50 methods. In future work we are going to investigate how to automatically generate the wrapper class.

[6] A transformation using XSLT always creates a new transformed XML document, the original document is not changed; i.e., XSLT does not perform in-place transformations.

Scientific Career

05/2005-05/2007
Darmstadt University of Technology
PhD student in the Software Technology Group of Prof. Mira Mezini

04/2002-04/2005
Darmstadt University of Technology
PhD student in the PhD Program Enabling Technologies for Electronic Commerce

10/1996-03/2002
Darmstadt University of Technology
Studies in Computer Science and Business Administration. Graduated as Diplom-Wirtschaftsinformatiker
(compareable to a joint master degree in computer science and business administration)

Bibliography

[AB01] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proceedings of the 13th Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2001.

[AE02] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002.

[AEK05] Raihan Al-Ekram and Kostas Kontogiannis. An xml-based framework for language neutral program representation and generic analysis. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2005.

[AGB03] Martin Auer, Bernhard Graser, and Stefan Biffl. A survey on the fitness of commercial software metric tools for service in heterogeneous environments: Common pitfalls. In *Proceedings of the Ninth International Software Metrics Symposium (METRICS)*. IEEE Computer Society, 2003.

[AH04] Cyrille Artho and Klaus Havelund. Applying jlint to space exploration software. In *Proceedings of Verification, Model Checking, and Abstract Interpretation: 5th International Conference*, volume 2937 of *Lecture Notes in Computer Science*. Springer, 2004.

[Ann06] Sun Microsystems. *Common Annotations for the Java Platform*, 2006. Specification Lead: Rajiv Mordani.

[ARK05] Jarallah S. Alghamdi, Raimi A. Rufai, and Sohel M. Khan. OOMeter: A software quality assurance tool. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2005.

[Asp06] Aspectj. `http://www.eclipse.org/aspectj/`, 2006.

[Bad00] Greg J. Badros. Javaml: a markup language for java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*. North-Holland Publishing Co., 2000.

[BAT06] The bytecode analysis toolkit (bat). `http://www.st.informatik.tu-darmstadt.de/BAT`, 2006.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10), 1996.

[BCE06] The bytecode engineering library (bcel). `http://jakarta.apache.org/bcel/manual.html`, 2006.

[BCF+05] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. Candidate recommendation, W3C, 2005. `www.w3.org/TR/xquery/`.

[BD02] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1), 2002.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1), 1998.

[BEN05] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve, version 5.5.0.6; open source (mixed-integer) linear programming system. `http://lpsolve.sourceforge.net/5.5/`, 2005.

[BHMO04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points.

In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD)*. ACM Press, 2004.

[BJL+03] Robert Balzer, Jens Jahnke, Marin Litoiu, Hausi A. Müller, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Ken Wong. 3rd international workshop on adoption-centric software engineering (acse). In *Proceedings of 25th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2003.

[BK01] Sarita Bassil and Rudolf K. Keller. Software visualization tools: Survey and analysis. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 2001.

[BKT+04] Bill Burke, Gavin King, Herve Tchepannou, Brian McSweeney, and James Cooley. xpetstore. http://xpetstore.sourceforge.net/, 2004.

[Blo01] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*. ACM Press, 2002.

[Bok99] Boris Bokowski. Coffeestrainer: statically-checked constraints on the definition and use of types in java. In *Proceedings of the 7th European software engineering conference (ESEC)*, volume 1687 of *Lecture Notes in Computer Science*. Springer, 1999.

[BPE05] OASIS. *Web Services Business Process Execution Language Version 2.0*, 2005. Committee Draft, 21th December.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7), 2000.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *Proceedings of the*

16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA). ACM Press, 2001.

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM Press, 2002.

[Bra04] Gilad Bracha. Pluggable type systems. In *Proceedings of the Workshop: Revival of Dynamic Languages (RDL)*, 2004.

[Bro83] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, 1983.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.

[BWDP00] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3), 2000.

[CAP05] Code analysis plug-in (CAP) 1.2.0. `http://cap.xore.de/`, 2005.

[CCS04] Gerardo Canfora, Luigi Cerulo, and Rita Scognamiglio. Measuring XML document similarity: a case study for evaluating information extraction systems. In *Proceedings of the 10th International Symposium on Software Metrics (METRICS)*. IEEE Computer Society, 2004.

[CD99] James Clark and Steve DeRose. XML path language (XPath) version 1.0. Recommendation, W3C, 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, and Robby Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software engineering (ICSE)*. ACM Press, 2000.

[CFKW95] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: a graphical navigator for software and document repositories. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1995.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 1994.

[CM04] Vasian Cepa and Mira Mezini. Declaring and enforcing dependencies between .net custom attributes. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3286 of *Lecture Notes in Computer Science*. Springer, 2004.

[CMM02] Michael L. Collard, Jonathan I. Maletic, and Andrian Marcus. Supporting document and data views of source code. In *Proceedings of the 2002 ACM symposium on Document engineering (DocEng)*. ACM Press, 2002.

[CNDE05] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. Incremental algorithms for interprocedural analysis of safety properties. In *Proceedings of 17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005.

[Cop06] Tim Copeland. Pmd. `http://pmd.sourceforge.net`, 2006.

[CPL06] Ilog cplex. `http://www.ilog.com/products/cplex/`, 2006.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the*

13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA). ACM Press, 1998.

[Cre97] Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages (DSL)*. USENIX, 1997.

[CRN03] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA)*. ACM Press, 2003.

[DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 1999.

[DEO+05] Sinisa Djukanovic, Sebastian Eifert, Matthias Orgler, Kai Stroh, Carole Urvoy, and Mario Vekic. Twomore. `http://www.pi.informatik.tu-darmstadt.de/se2004/byteme/`, 2005.

[DFK+04] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison Wesley, second edition, 2004.

[DL05] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31, 2005.

[DT03] Stéphane Ducasse and Sander Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(5), 2003.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*. ACM Press, 2003.

[EBGR99] Khaled El Emam, Sida Benlarbi, Nishith Goel, and Shesh Rai. A validation of object-oriented metrics. Technical Report NRC

43607, National Research Council Canada (Institute for Information Technology), 1999.

[ECCH01] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2001.

[Ecl05] EclipsePro Audit v4.2. `http://www.instantiations.com/eclipsepro/`, 2005.

[Ecl06] Eclipse 3.2. `http://www.eclipse.org`, 2006.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT)*. ACM Press, 1994.

[EGM+06] Michael Eichberg, Daniel Germanus, Mira Mezini, Lukas Mrokon, and Thorsten Schäfer. Qscope: an open, extensible framework for measuring software projects. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2006.

[EHMS05] Michael Eichberg, Michael Haupt, Mira Mezini, and Thorsten Schäfer. Comprehensive software understanding with sextant. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2005.

[Eic05] Michael Eichberg. Bat2xml: Xml-based java bytecode representation. In *Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode)*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[EJB03] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.1*, 2003. Specification Lead: Linda G. DeMichiel.

[EJB05] Sun Microsystems. *Enterprise JavaBeans Specification, Version 3.0 (Proposed Final Draft)*, 2005. Specification Lead: Linda G. DeMichiel and Michael Keith.

[EKMS06] Michael Eichberg, Sven Kloppenburg, Mira Mezini, and Tobias Schuh. Incremental confined types analysis. In *Proceedings of the Sixth Workshop on Language Descriptions, Tools and Applications (LDTA)*, volume 164 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2006.

[EKS+07] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. Automatic incrementalization of prolog based static analyses. In *Proceedings of the Ninth International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume (to appear) of *Lecture Notes in Computer Science*. Springer, 2007.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 2002.

[EM05] Michael Eichberg and Mira Mezini. Alice: Modularization of middleware using aspect-oriented programming. In *Proceedings of Software Engineering and Middleware: 4th International Workshop (SEM)*, volume 3437 of *Lecture Notes in Computer Science*. Springer, 2005.

[EMK+06] Michael Eichberg, Mira Mezini, Sven Kloppenburg, Klaus Ostermann, and Benjamin Rank. Integrating and scheduling an open set of static analyses. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2006.

[EMM01] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *The Journal of Systems and Software*, 56(1), 2001.

[EMO04] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In *Proceedings of Programming Languages and Systems: Second Asian Symposium (APLAS)*, volume 3302 of *Lecture Notes in Computer Science*. Springer, 2004.

[EMOS04] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schäfer. Xirc: A kernel for cross-artifact information

engineering in software development environments. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2004.

[EMS+04] Michael Eichberg, Mira Mezini, Thorsten Schäfer, Claus Beringer, and Karl-Matthias Hamel. Enforcing system-wide properties. In *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC)*. IEEE Computer Society, 2004.

[ESM05] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. Using annotations to check structural properties of classes. In *Proceedings of Fundamental Approaches to Software Engineering: 8th International Conference (FASE)*, volume 3442 of *Lecture Notes in Computer Science*. Springer, 2005.

[Fav01] Jean-Marie Favre. G^{SEE}: A generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 2001.

[Fav02] Jean-Marie Favre. A new approach to software exploration: Back-packing with g^{SEE}. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2002.

[FES03] J. Favre, J. Estublier, and R. Sanlaville. Tool adoption issues in a very large software company. Technical Report CMU/SEI-2003-SR-004, Carnegie Mellon — Software Engineering Institute, 2003.

[FL03] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. *ACM SIGPLAN Notices*, 38(11), 2003.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *ACM SIGPLAN Notices*, 37(5), 2002.

[FN99] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(3), 1999.

[FN01] Fabrizio Fioravanti and Paolo Nesi. A study on fault-proneness detection of object-oriented systems. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2001.

[Fon04] Philip W.L. Fong. Link-time enforcement of confined types for jvm bytecode. Technical Report CS-2004-12, Department of Computer Science, University of Regina, Regina, Saskatchewan, S4S 0A2, Canada, 2004.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[FP97] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics — A Rigorous & Practical Approach*. PWS Publishing, second edition, 1997.

[FT06] Paolo Falcarin and Marco Torchiano. Automated reasoning on aspects interactions. In *Proceedings of the 21th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2006.

[FvG03] Gregor Fischer and J. Wolff von Gudenberg. Simplifying source code analysis by an xml representation. *Softwaretechnik-Trends*, 23(2), 2003.

[FW04] David C. Fallside and Priscilla Walmsley. Xml schema. Recommendation, W3C, 2004. `www.w3.org/TR/xmlschema-0/`.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6), 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[GK02] Katsuhiko Gondow and Hayato Kawashima. Towards ansi c program slicing using xml. In *Proceedings of the Second Workshop*

on Language Descriptions, Tools and Applications (LDTA), volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[GKN02] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. AT&T, 2002. `http://www.graphviz.org/Documentation/dotguide.pdf`.

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*. ACM Press, 2001.

[GYF06] Emmanuel Geay, Eran Yahav, and Stephen Fink. Continuous code-quality assurance with safe. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM)*. ACM Press, 2006.

[Har05] Elliotte Rusty Harold. Zap bugs with pmd. `http://www-106.ibm.com/developerworks/java/library/j-pmd/`, January 2005.

[Har06] Elliotte Rusty Harold. Xom. `http://www.xom.nu/`, 2006.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI)*. ACM Press, 2002.

[HHR04] Daqing Hou, H. James Hoover, and Piotr Rudnicki. Specifying framework constraints with fcl. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 2004.

[HLW+92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *ACM SIGPLAN OOPS Messenger*, 3(2), 1992.

[HM06] Jason Hunter and Brett McLaughlin. Jdom 1.0. `http://www.jdom.org/`, 2006.

[Hog91] John Hogg. Islands: aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications (OOPSLA)*. ACM Press, 1991.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12), 2004.

[HR97] Mary Jean Harrold and Gregg Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, Ohio State University, 1997.

[HSvG03] Marbod Hopfner, Dietmar Seipel, and Jrgen Wolff von Gudenberg. Comprehending and visualizing software based on xmlrepresentations and call graphs. In *Proceedings of the 11 th IEEE International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 2003.

[HSvGF03] Marbod Hopfner, Dietmar Seipel, Jrgen Wolff von Gudenberg, and Gregor Fischer. Reasoning about source code in xml-representation. *Softwaretechnik-Trends*, 23(2), 2003.

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2006.

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. Gxl: Toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2000.

[Int06a] Intellij idea 5.1. `http://www.jetbrains.com`, 2006.

[Int06b] Interprolog. `http://www.declarativa.com/interprolog/`, 2006.

[JD03] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD)*. ACM Press, 2003.

[JDB06] Sun Microsystems. *JDBC 4.0 Specifiation*, 2006. Proposed Final Draft, Specification Lead: Lance Andersen.

[JMe99] JMetric version 1. `http://www.it.swin.edu.au/projects/jmetric/`, 1999.

[Joh79] Stephen C. Johnson. *Lint, a C Program Checker — UNIX Programmer's Manual*, 1979.

[JR00] Daniel Jackson and Martin Rinard. Software analysis: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*. ACM Press, 2000.

[JS99] Xiaoping Jia and Sotiris Skevoulis. A generic approach of static analysis for detecting runtime errors in java programs. In *Proceedings of the Twenty-Third Annual International Computer Software and Applications Conference (COMPSAC)*. IEEE Computer Society, 1999.

[KARW04] Konstantin Knizhnik, Cyrille Artho, Eric A. Raymond, and Mark Wutka. Jlint 3.0. `http://artho.com/jlint/`, 2004.

[Kay05a] Michael Kay. Saxon B 8.5.1. `http://saxon.sourceforge.net/`, 2005.

[Kay05b] Michael Kay. Saxon SA 8.5.1. `http://www.saxonica.com`, 2005.

[KC98] Rick Kazman and S. Jeromy Carrière. View extraction and view fusion in architectural understanding. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR)*. IEEE Computer Society, 1998.

[Kep04] Stephan Kepser. A simple proof for the turing-completeness of XSLT and XQuery. In *Proceedings of Extreme Markup Languages 2004*. Mulberry Technologies, 2004.

[KKL01] Hind Kabaili, Rudolf K. Keller, and Franois Lustman. Cohesion as changeability indicator in object-oriented systems. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th european conference on object-oriented programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[Koc04] Thorsten Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004.

[Lad03] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[Lan03] Michele Lanza. Codecrawler - lessons learned in building a software visualization tool. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2003.

[LD01] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*. ACM Press, 2001.

[Lim06] Limewire 4.10. `http://www.limewire.org`, 2006.

[Liv04] Benjamin Livshits. Finding security errors in java applications using lightweight static analysis. Annual Computer Security Applications Conference, Work-in-Progress Report, 2004.

[Liv05] Benjamin Livshits. Turning eclipse against itself: Finding bugs in eclipse code using lightweight static analysis. Eclipsecon '05 Research Exchange, 2005.

[LL05] Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the Usenix Security Symposium*. USENIX, 2005.

[LPLS87] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *The Journal of Systems and Software*, 7, 1987.

[LRY+04] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI)*. ACM Press, 2004.

[LS97] Claus Lewerentz and Frank Simon. Integrating an object-oriented metrics tool into sniff+. Technical Report I-22/1997, Technical University of Cottbus, 1997.

[LS98] Claus Lewerentz and Frank Simon. A product metrics tool integrated into a software development environment. In *Object-Oriented Technology — ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*. Springer, 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[MC04] Jonathan I. Maletic and Michael L. Collard. Supporting source code difference analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2004.

[McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993.

[MCK04] Jonathan I. Maletic, Michael Collard, and Huzefa Kagdi. Leveraging xml technologies in developing program analysis tools. In *Proceedings of the Fourth International Workshop on Adoption-Centric Software Engineering (ACSE)*. IET, 2004.

[MCM02] Jonathan I. Maletic, Michael L. Collard, and Andrian Marcus. Source code files as structured documents. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 2002.

[Mei05] Wolfgang Meier. exist 1.0. `http://exist.sourceforge.net/`, 2005.

[Met05] Metrics 1.3.6. `http://metrics.sourceforge.net/`, 2005.

[Mir04] Mircosoft. *PREfast with Driver-Specific Rules*, October 2004. `http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx`.

[Mit03] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[MK00] Evan Mamas and Kostas Kontogiannis. Towards portable source code representations using XML. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2000.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of the 20th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*. ACM Press, 2005.

[MMFA04] Nabor C. Mendonca, Paulo Henrique M. Maia, Leonardo A. Fonseca, and Rossana M. C. Andrade. Refax: A refactoring framework based on xml. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2004.

[MMG05] Cristina Marinescu, Radu Marinescu, and Tudor Gîrba. Towards a simplified implementation of object-oriented design metrics. In *Proceedings of the 11th IEEE International Symposium on Software Metrics (METRICS)*. IEEE Computer Society, 2005.

[MMN02] Gregory McArthur, John Mylopoulos, and Siu Kee Keith Ng. An extensible tool for source code representation using xml. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2002.

[MOD05] MODIS team at ISP RAS. Sedna. `http://modis.ispras.ru/sedna/`, 2005.

[MS95] Alberto Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software - Concepts & Tools*, 16(4), 1995.

[MSA+03] Tim Menzies, Justin S. Di Stefano, Kareem Ammar, Kenneth McGill, Pat Callis, Robert Chapman, and John Davis. When can we test less? In *Proceedings of the 9th International Symposium on Software Metrics (METRICS)*. IEEE Computer Society, 2003.

[MSCM02] Tim Menzies, Justin S. Di Stefano, Mike Chapman, and Kenneth McGill. Metrics that matter. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*. IEEE Computer Society, 2002.

[MTW93] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 1993.

[MW04] R. M. Marks and George Wilkie. Visualising object-oriented source code complexity using xml. In *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age (ICECCS)*. IEEE Computer Society, 2004.

[Net06] Netbeans 5.0. `http://www.netbeans.org`, 2006.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*. Springer, 1998.

[Par06] Terence Parr. Another tool for language recognition (antlr). `http://www.antlr.org/`, 2006.

[PBKM00] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 2000.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[PNB04] Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7), April 2004.

[RM02] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. ACM Press, 2002.

[RSK00] Sébastien Robitaille, Reinhard Schauer, and Rudolf K. Keller. Bridging program comprehension tools by design navigation. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2000.

[RSS+04a] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: smart analysis based error reduction. *ACM SIGSOFT Software Engineering Notes*, 29(4), 2004.

[RSS+04b] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Julian Dolby, Aaron Kershenbaum, and Larry Koved. Validating structural properties of nested objects. In *Companion to the 19th conference on Object-oriented programming systems, languages, and applications (OOPSLA)*. ACM Press, 2004.

[Sch92] Norman F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5), 1992.

[SCHC99] Susan Elliott Sim, Charles L. A. Clarke, Richard C. Holt, and Anthony Cox. Browsing and searching software architectures. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1999.

[Sco00] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.

[SDM06] SDMetrics version 2.0. `http://www.sdmetrics.com`, 2006.

[SEHM06] Thorsten Schäfer, Michael Eichberg, Michael Haupt, and Mira Mezini. The sextant software exploration tool. *IEEE Transactions on Software Engineering*, 32(9), 2006.

[SFM97] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. In *Proceedings of the Fifth International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 1997.

[Sle06] Sleepycat Software. Berkeley db xml. `http://www.sleepycat.com/`, 2006.

[SLL+88] Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11), 1988.

[SLVA97] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM, 1997.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the 11th International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1995.

[Sof05] Software AG. Tamino. `www.softwareag.com/tamino`, 2005.

[SP01] Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2001.

[SR06] Diptikalyan Saha and C.R. Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs. In *Proceedings of Practical Aspects of Declarative Languages: 8th International Symposium (PADL)*, volume 3819 of *Lecture Notes in Computer Science*. Springer, 2006.

[SSF06] Maximilian Störzer, Robin Sterr, and Florian Forster. Detecting precendence-related advice interference. In *Proceedings of the 21th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2006.

[SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2001.

[ST03] Hrvoje Simic and Marko Topolnik. Prospects of encoding java source code in xml. In *Proceedings of the 7th International Conference on Telecommunications (ConTEL)*. IEEE Computer Society, 2003.

[SWFM97] Margaret-Anne D. Storey, Kenny Wong, F. David Fracchia, and Hausi A. Müller. On integrating visualization techniques for effective software exploration. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*. IEEE Computer Society, 1997.

[SWM97] Margaret-Anne D. Storey, K. Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs? In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 1997.

[SWM00] Margaret-Anne D. Storey, K. Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3), 2000.

[SY02] Mati Shomrat and Amiram Yehudai. Obvious or not? regulating architectural decisions using aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development (AOSD)*. ACM Press, 2002.

[TAT06] TATA consultancy services. Assent. `http://www.tcs.com/0_products/assent/`, 2006.

[TCSD04] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Ignatios Deligiannis. Probabilistic evaluation of object-oriented systems. In *Proceedings of the 10th International Symposium on Software Metrics (METRICS)*. IEEE Computer Society, 2004.

[Tog05] Borland together architect 2006 / designer 2006. `http://www.borland.com`, 2005.

[VB01] Jan Vitek and Boris Bokowski. Confined types in java. *Software Practice and Experience*, 31(6), 2001.

[vGB02] Jilles van Gurp and Jan Bosch. Design erosion: Problems and causes. *Journal of Systems and Software*, 61(2), 2002.

[Vla06] Pavel Vlasov. Hammurapi 4.0. `http://hammurapi.org`, 2006.

[vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8), 1995.

[Vol06] Nic Volanschi. A protable compiler-integrated approach to permanent checking. In *Proceedings of the 21th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2006.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction (CC)*, volume 1781 of *Lecture Notes in Computer Science*. Springer, 2000.

[Web05] Sun Microsystems. *Web Services Metadata for the Java Platform*, 2005. Version 2.0, Specification Lead: Stuart Edmondston and Brian Zotter.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE Computer Society, 1998.

[XSB06] Xsb prolog 2.7.1. `http://xsb.sourceforge.net/`, 2006.

[YSM02] Ping Yu, Tarja Systä, and Hausi Müller. Predicting fault-proneness using oo metrics — an industrial case study. In *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2002.

[ZPV03] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight java. *ACM SIGPLAN Notices*, 38(11), 2003.