**Workshop Proceedings**

Fir0002/Flagstaffotos
GFDL, Version 1.2 only
http://www.gnu.org/licenses/fdl-1.2.txt

# Third International Workshop on Academic Software Development Tools and Techniques

Co-located with the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)

September 20, 2010
Antwerp, Belgium

H. Kienle (Ed.)

# Committees

### Organizing Committee

Mark van den Brand, TU/Eindhoven, The Netherlands (primary contact)
Kim Mens, Université catholique de Louvain (UCL), Belgium
Holger Kienle, University of Victoria, Canada & Mälardalen University, Sweden
Anthony Cleve, INRIA Lille, France

### Program Committee

Gabriela Arevalo, Universidad Austral & CONICET, Buenos Aires, Argentina
Emilie Balland, INRIA Bordeaux, France
Martin Bravenboer, LogicBlox, USA
Loek Cleophas, University of Pretoria, South Africa & Eindhoven University of Technology, The Netherlands
Francisco Durán, University of Málaga, Spain
Alexander Egyed, Johannes Kepler University, Austria
Rudolf Ferenc, University of Szeged, Hungary
Robert Fuhrer, IBM, USA
Yann-Gaël Guéhéneuc, École Polytechnique de Montréal, Canada
Gorel Hedin, Lund University, Sweden
Robert Hirschfeld, Hasso-Plattner-Institute at the University of Potsdam, Germany
Andy Kellens, Vrije Universiteit Brussel, Belgium
Rainer Koschke, Universität Bremen, Germany
Adrian Kuhn, University of Bern, Switzerland
Mircea Lungu, University of Lugano, Switzerland
Franco Mazzanti, CNR, Italy
Pierre-Etienne Moreau, INRIA Nancy, France
Gail Murphy, University of British Columbia, Canada
Terence Parr, University of San Francisco, USA
Romain Robbes, University of Chile Santiago, Chile
Tony Sloane, Macquarie University Sydney, Australia
Gabriele Taentzer, Philipps-Universität Marburg, Germany
Alexandru C. Telea, University of Groningen, The Netherlands
Jurgen Vinju, CWI, The Netherlands
Markus Voelter, itemis AG, Germany
Roel Wuyts, IMEC, Belgium & KULeuven, Belgium

### Additional Reviewers

Alexandre Bergel, University of Chile Santiago, Chile
Paul Brauner, Rice University, Houston, TX

## Aims and Topics of the WASDeTT Series

The *Workshop on Academic Software Development Tools and Techniques* (WASDeTT) series is motivated by the observation that tools and tool building play an important role in applied academic software engineering research. The tangible results of research projects are often embodied in a tool. Even though tool building is a popular technique to validate research (e.g., proof-of-concept prototyping followed by user studies), it is neither simple nor cheap to accomplish. Furthermore, researchers hardly get credits for the effort invested in building these tools. Given the importance of tool building and the significant cost associated with it, our workshop allows interested researchers to share their tool building experiences and to explore how tools can be build more effectively and efficiently. It also allows other researchers to benefit and learn from the work of others.

The purpose of this workshop is not to focus on any specific kind of these tools (say, refactoring or program comprehension tools) or architecture (say, Eclipse or IMP). We want to gather researchers working on different tools, with the goal of providing a forum where tool builders can talk about common issues relevant to all tool builders, and builders of academic research prototypes in particular, such as

- Should tool building remain a craft?
- Should academic tools be of commercial quality?
- How to integrate and combine independently developed tools?
- What are the positive lessons learned in building tools?
- What are the pitfalls in tool building?
- What are the good practices?
- What are effective techniques to improve the quality of academic tools?
- What is needed to build an active community of developers and users?
- Are there any useful tool building patterns for software engineering tools?
- How to compare or benchmark such tools?
- What particular languages and language paradigms are suited to build software engineering tools?

WASDeTT-3 is associated with an *Experimental Software and Toolkits* (EST) special issue to be published in Elsevier's Science of Computer Programming (SCP). All workshop papers are potential candidates for the special issue. The authors of a subset of these articles will be invited to submit to EST based on the quality of their paper and their tool presentation during the workshop. Both the paper and the tool will be reviewed and if accepted for publication, SCP will publish not only the paper but also the tool.

## Accepted Papers

**Sourcerer – An Infrastructure for Large-scale Collection and Analysis of Open-source Code**
Sushil Bajracharya, Joel Ossher and Cristina Lopes ............................................................................1

**SyLaGen: From Academic Tool Engineering Requirements to a new Model-based Development Approach**
Moritz Balz, Michael Striewe and Michael Goedicke ..........................................................................32

**Industrialization of Research Tools: the ATL Case**
Hugo Brunelière, Jordi Cabot, Frédéric Jouault, Massimo Tisi and Jean Bezivin ............................44

**Disnix: A toolset for distributed deployment**
Sander van der Burg and Eelco Dolstra ..............................................................................................58

**A Proof Repository for Formal Verification of Software**
Michael Franssen ..................................................................................................................................76

**Processes and Practices for Quality Scientific Software Projects**
Veit Hoffmann, Horst Lichter and Alexander Nyßen ..........................................................................95

**From Design to Tools: Process Modeling and Enactment with PDE and PET**
Marco Kuhrmann, Georg Kalus, Manuel Then and Eugen Wachtel ..................................................109

**A Platform for Experimenting with Language Constructs for Modularizing Crosscutting Concerns**
Tim Molderez, Hans Schippers, Dirk Janssens, Michael Haupt and Robert Hirschfeld .................129

**A Visual Analytics Toolset for Program Structure, Metrics, and Evolution Comprehension**
Dennie Reniers, Lucian Voinea, Ozan Ersoy and Alexandru Telea ..................................................146

**Building industry-ready tools: FAMA Framework & ADA**
Pablo Trinidad, Carlos Müller, Jesús García-Galán and Antonio Ruiz-Cortés ...............................160

**Developing A Generic Debugger for Advanced-Dispatching Languages**
Haihan Yin and Christoph Bockisch ...................................................................................................174

# Sourcerer - An Infrastructure for Large-scale Collection and Analysis of Open-source Code

Sushil K Bajracharya, Joel Ossher, Cristina V Lopes

*University of California Irvine, USA*

**Abstract**

A large amount of open source code is now available online, presenting a great potential resource for software developers. This has motivated software engineering researchers to develop tools and techniques to allow developers to reap the benefits of these billions of lines of source code available online. However, collecting and analyzing such a large quantity of source code presents a number of challenges. Although the current generation of open source code search engines provide access to the source code in an aggregated repository, they generally fail to take advantage of the rich structural information contained in the code they index. This makes them significantly less useful than Sourcerer for building state-of-the art software engineering tools, as these tools often require access to both the structural and textual information available in source code.

We have developed Sourcerer, an infrastructure for large-scale collection and analysis of open source code. By taking full advantage of the structural information extracted from source code in its repository, Sourcerer provides a foundation upon which state of the art search engines and related tools can easily be built. We describe the Sourcerer infrastructure, present the applications that we have built on top of it, and discuss how existing tools could benefit from using Sourcerer.

*Keywords:* Open Source, Internet-scale code retrieval, Data Mining, Sourcerer, Static Analysis, Software Information Retrieval

## 1. Introduction

The popularity of the Open Source Software movement has dramatically increased the general availability of free, high quality source code. The repositories that host open source software are growing at an exponential rate [30], and the software itself is seeing increasing usage, from both developers and the general public. With respect to developers, open source software is often used as part of the development infrastructure, as well as in the code itself, in the form of reusable libraries and frameworks [35, 5]. This growth in the usage and availability of open source software provides a rich opportunity for the creation of novel software engineering solutions. However, collecting, analyzing, and actually using such large quantities of source code is quite challenging, which makes building and evaluating any new techniques significantly difficult. In

*Email addresses:* sbajrach@ics.uci.edu (Sushil K Bajracharya), jossher@ics.uci.edu (Joel Ossher), lopes@ics.uci.edu (Cristina V Lopes)

this paper we present Sourcerer, an infrastructure developed to tackle these key challenges. The paper provides details on the models Sourcerer uses to represent open source code, the content Sourcerer stores, the tools that comprise Sourcerer, and the services that Sourcerer provides. The current version of Sourcerer is designed to work with open source projects developed using Java.

The proliferation of open source software has given rise to two recent trends in the software industry and the academic software engineering research community.

1. **Commercial code search engines**. The growth in the amount of open-source software in public repositories is reflected in the emergence of several commercial code search engines, such as Koders, Krugle and Google Code Search [67, 68, 11]. These code search engines allow developers to use a single site to search billions of lines of source code collected from various repositories on the Web.
2. **Research trends in leveraging large software repositories**. There has been considerable research effort in building specialized search tools for software developers. Studies of developers' activities reveal a routine use of web resources during development, in particular code examples and snippets [24, 36]. Supporting these observations several tools have been proposed that utilize code retrieved from repositories. Examples include recommending APIs [37], code-completion [25], finding reusable code fragments [41], finding and synthesizing API examples [36, 26], and finding application for prototyping [34].

The commercial code search engines go a fair way towards fulfilling software developers' needs for finding relevant open source code. However, their performance is often lacking, as any user can attest. As a result, the software engineering research community has created a number of novel approaches for large-scale code retrieval. A common theme among these approaches is the use of structural information along with the more traditional textual information extracted from the code. However, conducting research that requires this rich structural information presents three major challenges:

1. **Collection:** The primary challenge in collecting source code off the Internet is that there is no standard method of distribution. Open source projects are generally hosted by large open source repositories, such as Sourceforge [69], Google Code Hosting [66], and Apache [2], which rarely provide hooks for performing this type of collection. Ultimately, the best way to obtain the source code is to scrape the download links and version control systems from the project web pages. However, the use of different version control systems, download protocols, and constantly changing format/content of the web pages makes automating this collection process rather tedious.
2. **Analysis:** In order to leverage the structural information available in source code, one has to be able to first extract the information itself. In order to fully extract structural information from source code, the code must be declaratively complete; i.e. all the dependencies must be resolved. Unfortunately, when it comes to source code from the internet, there is no guarantee that the code is declaratively complete; missing dependencies and incomplete files are quite common. Scaling the analysis to thousands of projects further complicates matters, as it eliminates the feasibility of performing any type of manual processing.
3. **Application:** Due to the challenges posed by collection and analysis, it is often impossible to rapidly design and evaluate applications that make use of both textual and structural information extracted from large quantities of source code. The upfront cost of constructing a repository and implementing sound analysis tools makes research in large scale code retrieval non-trivial.

The Sourcerer project explores various applications of the large-scale collection and analysis of open source code. We have developed a research infrastructure that is principally driven by the need to tackle the three major challenges mentioned above. In summary, the Sourcerer infrastructure enables the following:

1. **Collection** of large amount of source code from open source repositories to build a reference repository for research in large scale source code analysis.
2. Automated **analysis** of arbitrary Java source code that exists out in the wild (on the internet).
3. Rapid development and evaluation of **applications** that leverage large amount of preprocessed, cross-linked and aggregated repository of source code.

This paper provides details on important aspects of Sourcerer that enable the above features. The paper is an extension of our earlier publications on Sourcerer [20, 61]. It includes more detailed description on the latest version of the Sourcerer infrastructure and makes the following contributions:

- Provides key details on the design and implementation of the Sourcerer infrastructure that enables collection and analysis of large amount of open source code.
- A summary of applications enabled by Sourcerer's infrastructure and a list of its major contributions in the area of large-scale code retrieval and source code data mining.
- Key details on implementation, configuration, and availability of Sourcerer to motivate its adoption by external researchers.

The Sourcerer Infrastructure primarily consists of five different components:

1. **Models:** A set of abstractions that capture the elements of textual and structural information in Sourcerer's source code collection.
2. **Tools:** A collection of loosely coupled tools responsible for the collection and analysis of source code.
3. **Stored Content:** Artifacts that are produced using various tools and stored locally in Sourcerer's repository.
4. **Services:** A layer of abstraction that enables access to the Stored Content.
5. **Applications:** that leverage the Stored Content accessed via various Services.

**Paper Organization:** The paper begins by covering the five components that constitute the Sourcerer infrastructure. Sections 2 through 5 capture how we built the infrastructure by describing the models, tools, and services. In Section 6, we discuss the applications built using Sourcerer, which validates the applicability and impact of the Sourcerer infrastructure. Section 7 provides information on availability of the Sourcerer infrastructure for use by other researchers. In Sections 8 and 9 we discuss related and future work, and cover how Sourcerer relates to existing similar tools and platforms. We conclude in Section 10. The Appendix contains details on implementation to motivate external users to obtain, use, and extend the Sourcerer's open source infrastructure.

## 2. Models

Three models define the basic mechanisms for storing and retrieving information from the source code available in Sourerer's repository.

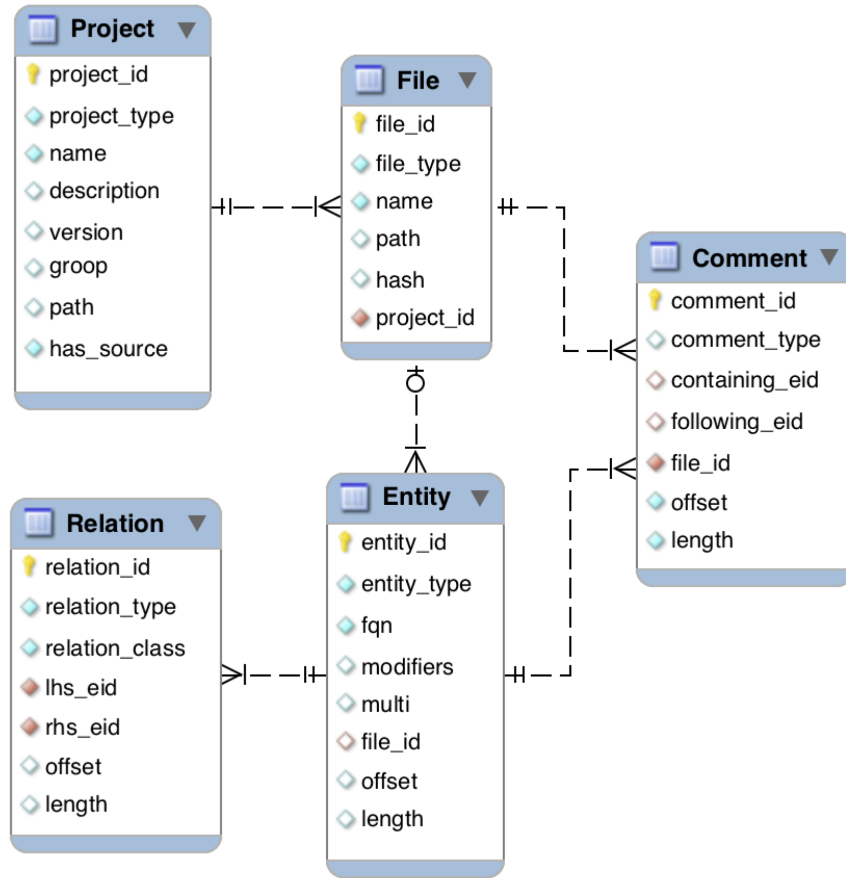



Figure 1: Sourcerer's Relational Model

### *2.1. Storage Model*

The Storage Model captures the layout and structure of the physical files in Sourcerer's local repository. A layered directory structure was chosen for two main reasons. First, it allows projects from the same source to be grouped together, which makes adding or removing content more straightforward. Second, after initially implementing a flat version, we discovered that the file system did not adequately handle having tens of thousands of subdirectories in a single directory. The files collected from open source projects are stored in a folder according to the following template:

```
<repo_root>/<batch>/<id>
```

Above, `<repo_root>` is a folder assigned as the root of Sourcerer's file repository. Given the root folder, the individual project files are stored in a two-level directory structure defined by the path fragment `<batch>/<id>`. Each `<batch>` folder contains a semi-arbitrary collection of projects. For example, a batch could be a crawl from a specific online repository or a collection of fixed number of projects. Inside `<batch>`, another set of folders exist. Each second-level

folder in the local repository, indicated by `<id>` in the above template, contains the contents of a specific project. Each `<id>` directory contains a single file and two sub-directories, as shown below:

```
<repo_root>/<batch>/<id>/project.properties
<repo_root>/<batch>/<id>/download/
<repo_root>/<batch>/<id>/content/
```

Above, `project.properties` is a text file that stores the project metadata as a list of name value pairs. The `download` folder contains the compressed file packages that were fetched from the originating repository (e.g. a project's distribution in Sourceforge). The `content` directory contains the expanded contents of the downloads directory. Once the contents of the download directory have been expanded, the directory itself is usually emptied in order to free up space.

The project contents in the `content` directory can take two different forms, depending on its format in the initial repository. If the project contents are checked out from a remote software configuration management systems (such as svn and cvs), the file located at a relative path `path` in the originating repository (e.g Sourceforge) exists in Sourcerer's file repository at the following absolute path:

```
<repo_root>/<batch>/<id>/contents/<path>
```

If, instead, the project is fetched from a package distribution, a source file can be found in Sourcerer's file repository at the following absolute path:

```
<repo_root>/<batch>/<id>/contents
      /package.<i>/<path>
```

Above, `package.<i>` indicates a unique folder for each $i^{th}$ package that is found in a remote repository. `path` indicates a relative path of a source code file that is found inside the $i^{th}$ archived package, which is unarchived inside the `package.<i>` folder.

**Project metadata:** The `project.properties` file is a generic project description format that generalizes the project metadata from the online repositories. Many of the attributes in `project.properties` are optional, except for the following:

- `crawledDate`: indicates when the crawler picked up the project information
- `originRepositoryUrl`: URL of the originating repository; e.g. http://sourceforge.net
- `name`: project's name as given in the originating repository
- `containerUrl` : project's unique URL in the originating repository

And, one or both of the following: (i) Information on project's software configuration management (SCM) system indicated by `scmUrl` (ii) Information on project's source package distributed on the originating repository:

- `package.size` indicating total number of packages distributed
- `package.name.`*i* indicating name of the $i^{th}$ package, where $1 <= i <= size$, and $i$ indicates a unique integer denoting a package number.

- `package.sourceUrl.`*i* indicating the URL to get the $i^{th}$ package from the originating repository.

Two sample `project.properties` files showing metadata for two projects is given in Appendix A.

**Jar Storage**: In addition to the top-level `batch` directories described above, the local repository also contains a single `jars` directory. The jars directory is structured as follows:

```
<repo_root>/jars/project/<jar_path>
<repo_root>/jars/maven/<jar_path>
<repo_root>/jars/index.txt
```

The `project` subdirectory contains all of the jar files that come packaged with the projects in the main repository. This directory is populated by crawling through the repository itself, and copying every jar found. The copying is done so that these jar files can be modified, if necessary, without altering the original projects. The `maven` subdirectory contains a mirror of the Maven2 Central Repository. Lastly, `index.txt` contains an index that maps from the MD5 hash of a jar file to its location in the directory structure. This index is used to link the jar files from the projects to the files contained in the `jars` directory.

The Storage Model provides a standard for storing project files in Sourcerer and is not directly used by applications. The description above will be useful for those interesting in downloading and using the Sourcerer's reference file repository from [54]. Applications rely on other higher-level abstractions to access the contents stored in Sourcerer.

*2.2. Relational Model*

Sourcerer's Relational Model defines the basic source code elements and the relations between those elements. The metamodel is specific to Java, and is designed to capture its latest version. We decided to go with a language-specific metamodel, rather than a more generic metamodel such as FAMIX [29], as it allows us to more concisely represent Java-specific features. Our model supports a fine-grained representation of the structural information extracted from source code. It also links the code elements/relations with their locations in physical artifacts.

Two major goals guided the design of Sourcerer's relational model. First, it had to be sufficiently expressive as to allow fine-grained search and structure-based analyses. Second, it had to be efficient and scalable enough to include the large amount of code from thousands of open source projects. To meet these two goals we decided to use an adapted version of Chen et al.'s [27] C++ entity-relationship-based metamodel as Sourcerer's relational model for source code. In particular, their decision to focus on what they termed a *top-level declaration* granularity provides a good compromise between the excessive size of finer granularities and the analysis limitations of coarser ones.

Our metamodel has evolved two primary ways from its original version. First, it was adapted to include the features introduced by Java 1.5. While adding these features contributed a fair amount of complexity to the metamodel, we felt that it was necessary given the increasing prevalence of open source code using the features. Second, after receiving some feedback from users, we decided to add local variables to our metamodel, allowing for a slightly finer granularity in our analysis.

The relational model consists of the following five elements: Project, File, Entity, Comment, and Relation.

| PACKAGE |
| --- |
| CLASS<br>INTERFACE<br>ENUM<br>ANNOTATION<br>INITIALIZER<br>FIELD<br>ENUM CONSTANT<br>CONSTRUCTOR<br>METHOD<br>ANNOTATION ELEMENT<br>PARAMETER<br>LOCAL VARIABLE |
| PRIMITIVE<br>ARRAY<br>TYPE VARIABLE<br>WILDCARD<br>PARAMETRIZED TYPE |
| UNKNOWN |

Table 1: Entity Types

A **project** model element exists for every project contained in Sourcerer's repository, as well as every unique Jar file. A project therefore contains either a collection of Java source files and jar files, or a collection of class files. A **file** model element represents these three types of files: source (`.java`), jar (`.jar`) or class (`.class`). Both source and class files are linked to sets of **Entities** contained within them, and to the **Relations** that have these entities as their source and target. Jar files, on the other hand, are linked to their corresponding jar projects, which in turn contains all of the **entities** and **relations**.

An **entity** model element either corresponds to an explicit declaration in the source code (e.g. Class, Interface, Method etc), a Java package[1], or Java types that are used but do not correspond to a known explicitly declared type (e.g. Array, Type Variable). An entity type is `UNKNOWN` when the type cannot be determined due to uncertainty in the analysis. Table 1 lists all entity model element types defined in Sourcerer. These types all adhere their standard meaning in Java, as defined in the Java Language Specification (JLS) [32].

A **relation** model element represents a dependency between two Entities. A dependency $d$ originating from a source entity $s$ to a target entity $t$ is stored as a Relation $r$ from $s$ to $t$. Table 2 contains a complete list of the relation types with a brief description and example for each. All of the relations are binary, linking a source entity to a target. The source entity for a relation is smallest entity that contains the code that triggers that relation. While containment is clear for most of the entities, it should be noted that `FIELD`s are considered to contain their initializer code and `ENUM CONSTANT`s are considered to call their constructors. The source entity is always found within the project being examined. This is not necessarily true of the target entity. It can be a reference to the Java Standard Library or any other external jar. In fact, sometimes it is impossible to resolve the type of the target entity, due to missing dependencies.

A **comment** model element represents the comments defined in the Java source code.

[1]Packages are not considered to be standard declared entities as they do not have a single declaration

| Relation | Description | Example |
|---|---|---|
| INSIDE | Physical containment | java.lang.String INSIDE java.lang |
| EXTENDS | Class extension | java.util.LinkedList EXTENDS java.util.AbstractSequentialList |
| IMPLEMENTS | Interface implementation | java.util.LinkedList IMPLEMENTS java.util.List |
| | Interface extension | java.util.List IMPLEMENTS java.util.Collection |
| HOLDS | Field type | java.lang.String.offset HOLDS int |
| RETURNS | Method return type | java.lang.String.toCharArray() RETURNS char[] |
| READS | Field read | ...String.<init>(java.lang.String) READS java.lang.String.offset |
| WRITES | Field write | java.lang.String.<init>() WRITES java.lang.String.offset |
| CALLS | Method invocation | ...String.indexOf(int) CALLS java.lang.String.indexOf(int,int) |
| INSTANTIATES | Constructor invocation | foo() INSTANTIATES java.lang.String.<init> |
| THROWS | Declared checked exception | java.io.Writer.write(int) THROWS java.io.IOException |
| CASTS | A cast expression | java.langString.equals(java.lang.Object) CASTS java.lang.String |
| CHECKS | An instanceof expression | java.langString.equals(java.lang.Object) CHECKS java.lang.String |
| ANNOTATED BY | Annotation | java.lang.Override ANNOTATED BY java.lang.annotation.Target |
| USES | Any reference | java.lang.String.<init>() USES char |
| HAS ELEMENTS OF | Array element type | char[] HAS ELEMENTS OF char |
| PARAMETRIZED BY | Associated type variables | java.util.List PARAMETRIZED BY <E> |
| HAS BASE TYPE | Generic base type | java.util.List<java.lang.String> HAS BASE TYPE java.util.List |
| HAS TYPE ARGUMENT | Generic type argument | java.util.List<java.lang.String> HAS TYPE ARGUMENT java.lang.String |
| HAS UPPER BOUND | ? extends TYPE | <? extends java.util.List> HAS UPPER BOUND java.util.List |
| HAS LOWER BOUND | ? super TYPE | <? super java.util.List> HAS LOWER BOUND java.util.List |

Table 2: Relation Types

Figure 2 shows Sourcerer's relational model using an ER-diagram. It shows the five elements of Sourcerer's relational model and a set of attribute for each of them. Table 3 provides the details on all the attributes of the model elements. Figure 2 and Table 3 provide information on how the model elements are linked with each other, and how the attributes in the relational model link the relational model elements with the storage model. For example, Project element's 'path' attribute links it to the physical location defined by the storage model.

Various tools in Sourcerer make use of this information to connect the relational information with the textual content stored in the physical files.

### *2.3. Index Model*

The Index Model complements Sourcerer's relational model by facilitating the application of information retrieval techniques to code entities. Sourcerer's information retrieval component is based on the popular Lucene [6] information retrieval engine, and therefore our index model follows Lucene's general approach. More details on Lucene's content model is available in [59].

Our index model matches a Lucene **document** to each entity in the relational model. A document is made up of a collection of **field**s, each field being a name/value pair. The simplest form of value is a collection of **term**s, where a term is the basic unit for search/retrieval. Terms are extracted from various parts of an entity, and stored in the fields of the document corresponding to that entity.

Fields in Sourcerer's index models can be categorized into five types:

1. Fields for *basic retrieval* that store terms coming from various parts of a code Entity
2. Fields for *retrieval with signatures* that store terms coming from method signatures and also terms that indicate number of arguments a method has
3. Fields storing *metadata*, for example the type of the Entity, so that a search could be limited to one or more types of entities
4. Fields that store information to facilitate *retrieval based on structural similarity* (e.g. fields storing fully qualified names (FQNs) of used entities and terms extracted from similar entities
5. Fields that pertain to some *metric* computed on an entity
6. Fields that store ids of entities for *navigational/browsing queries*

Being based on Lucene, Sourcerer's index model is quite flexible. Depending on a specific search application, an instance of a Sourcerer's index schema can have a subset of various field types listed above. Appendix B shows an example code index schema used in two of Sourcerer's search applications: Sourcerer Code Search engine [18] and CodeGenie [46, 44, 45, 47].

Table 4 presents a subset of the fields available in the Sourcerer index. Sourcerer's search index can be searched using Lucene's query language. The following Lucene query demonstrates how different fields are utilized to express a query that incorporates textual as well as structural information:

```
short_name: (week date)
  AND entity_type: METHOD
  AND m_ret_type_sname_contents: String
  AND m_args_fqn_contents: Date
```

The above query has the following meaning: find a method with the terms `week` and `date` in its short name (or simple name in JLS [32]), that returns a type with short name `String` and takes in an argument type with the term `Date` in its name.

| | **Description** |
|---|---|
| **Project** | |
| project_id | unique identifier for a project |
| project_type | denotes whether this project represents a crawled project, or a Jar file |
| name | name of the project as it appears in the originating Internet repository |
| description | description of the project from the originating Internet repository |
| version | version of this project as extracted from originating Internet repository |
| groop | specific field applicable to Maven Jars |
| path | corresponds to the `<batch>/<id>` path fragment as defined by the storage model |
| has_source | denotes whether the project contains source files |
| **File** | |
| file_id | unique identifier for a file |
| file_type | denotes the file's type - source, Jar, class |
| name | name of the file in the file system |
| path | corresponds to either `<batch>/<id>/contents/<path>`, or `jars/<jar_path>` as defined by the storage model |
| hash | unique MD5 hash, applicable for Jars only |
| project_id | project_id that this file belongs to |
| **Entity** | |
| entity_id | unique identifier for an Entity |
| entity_type | one of the several code entity types. E.g. CLASS, METHOD etc |
| fqn | Fully qualified name (FQN) of the entity |
| modifiers | modifiers defined for the code entity |
| multi | denotes array dimension, applicable for ARRAY types only |
| file_id | file_id that this entity is extracted from |
| offset | start position of this entity in the source file |
| length | length of this entity in the text (source file) |
| **Relation** | |
| relation_id | unique identifier for a relation |
| relation_type | one of the several code relation types. E.g. CALLS, EXTENDS etc |
| relation_class | denotes whether the relation terminates to a library or a local entity |
| lhs_eid | the source entity that the relation originates from |
| rhs_eid | the target entity that the relation terminates into |
| offset | start position in the source entity's corresponding file where this relation exists |
| length | length of the text in source code where this relation spans |
| **Comment** | |
| comment_id | unique identifier for a comment |
| comment_type | denotes the comment's type - Javadoc, Block, Line |
| containing_eid | the immediate code entity that contains this comment |
| following_eid | the immediate code entity that follows this comment |
| file_id | file where this comment is found |
| offset | start position of comment in the source file |
| length | length of this comment in text (source file) |

Table 3: Sourcerer's Relational Model Elements Details

| Index Field | Description |
|---|---|
| *Fields for basic retrieval* | |
| fqn_contents | Tokenized terms from the FQN of an entity |
| short_name | right most fragment of the FQN (w/o method arguments for methods) |
| *Fields for retrieval with signatures* | |
| m_args_fqn_contents | method's formal arguments tokenized into terms after passing each of them through keyword extractor |
| m_ret_type_sname_contents | short name of the method's return type tokenized |
| *Fields Storing metadata* | |
| entity_type | String representation of entity type. Eg; "CLASS" |
| *Fields for navigation* | |
| fan_in_mcall_local | entity ids of all local callers for a method from the same project |

Table 4: Sample search index fields

Our recent publication [21] provides a more elaborate discussion on how Sourcerer's index model is used in a code-retrieval application.

## 3. Stored Content

The Sourcerer infrastructure maintains a collection of stored content corresponding to each of the three models.

A **File Repository** keeps a collection of files downloaded and fetched from open source repositories in the Internet. The structure of the code repository follows the storage model.

Two different databases store the relational information about the contents in the file repository. First, **ArtifactDB** stores limited information about the jar files found in the repository in order to enable the automated resolution of missing dependencies [62]. Second, **SourcererDB** stores the relational information on all projects, files and code entities that exist in the code repository. Both of the code databases exist as MySql databases whose schemas confirm to Sourcerer's relational model.

A Lucene based **Search Index** is available that stores information about terms extracted from each code entity in the corresponding documents and fields. The search index uses a code index schema following the index model.

## 4. Services

All the artifacts managed and stored in Sourcerer are accessible through a set of web services. These services provide a layer of abstraction and programmatic access to rapidly build applications that can leverage the underlying content stored in Sourcerer.

**Relational Query:** Both ArtifactDB and SourcererDB are implemented as MySql databases. They provide direct access to query the underlying structural/relational information in Sourcerer using standard SQL.

**Repository Access:** This service provides access to the textual content of four of Sourcerer's model elements: files, entities, relations and comments. Repository access is a simple HTTP-based web service that returns the full text when given a unique id.

**Dependency Slicing:** This service provides dependency slices of the code entities in SourcererDB. A dependency slice of an entity is a program (collection of Java source files) which includes that entity as well as all the entities upon which it depends. Requested slices are packaged into zip files, and should immediately be compilable. The dependency slicing service can take in one or more entity ids and return a zip file containing the collection of sliced/synthesized Java files that the given set of entities depend on.

**Code Search:** This service implements a query processing and retrieval facility. Client applications (such as CodeGenie [46, 44, 45, 47]) can send queries as a combination of terms and fields and the service returns a result set with detailed information on the entities that matched the queries. The query language is based on Lucene's implementation using which clients can express structural information in the queries. The current code search service is implemented as a customization of the Solr search engine [15]. Solr is a front end for Lucene, and supports a wide range of retrieval functions such as: basic retrieval of code entities based on the vector space model, faceting based on fields defined in the code index schema, similarity searches etc. All these features are available over Sourcerer's code search index. The matching and scoring (ranking) of entities follow Lucene's implementation. Further details on how Lucene/Solr match the query terms in index fields and scores the matched entities is available from our previous publication [21]; a more definitive source is [10]. In summary, a boolean retrieval is performed based on a Lucene query as shown in Section 2.3, then all matched entities (documents) are ranked using the TF-IDF measure [58].

**Similarity Calculation:** The Similarity Calculation service takes in an entity_id of an entity 'e' and returns a list of other entities that are similar to 'e'. Currently, the similarity calculator can suggest similar entities based on three different measures of usage similarity. For this purpose, the similarity calculator uses the usage information stored in SourcererDB. The Structural Semantic Indexing technique that we presented in [21] makes use of the similarity calculation service. Further details on similarity calculation is available in [21].

Except the Relational Query service, all other services are simple HTTP based services. Currently three services are open to the public. A detailed description of how to use these services is available online [40].

## 5. Tools

A number of loosely coupled tools are available in the Sourcerer infrastructure. These tools are primarily responsible for collecting and analyzing source code and producing Sourcerer's stored contents.

**Code Crawler:** Sourcerer consists of a multithreaded plugin-based code crawler that can crawl the web pages in online source code repositories. To adapt with the changes and differences with web pages in different Internet repositories, the crawler follows a plugin based design. A separate plugin can be written targeting the crawl of a repository. This makes it possible to just update the plugin or add new plugins when a different or new web site has to be crawled. Currently the crawler consists of plugins for Sourceforge [69], Java.net [3], Tigris [4], Google Code

Hosting [66], and Apache [2]. The crawler takes a set of root URLs as an input and produces a list of download URLs and version control links along with other project specific metadata. This project specific metadata is in the form as specified by (the `project.properties` file in) the storage model.

**Repository Creator:** The Repository Creator tool is responsible for parsing the output list from the Code Crawler, filtering noise from the list (e.g. removing duplicate links), and downloading the contents from the online repositories to Sourcerer's local file repository. Given a local file repository's root folder, the repository creator creates the required folder structure and places the contents as specified by Sourcerer's storage model. The repository creator first creates the two level folder structure based on the number of projects it needs to add to the repository. Second, it creates the `project.properties` file describing each project. Third, it fetches the files from remote/original repositories. `project.properties` has metadata about two content sources in remote repositories: (i) Source Configuration Management systems such as svn and cvs, and (ii) downloadable packages such as compressed distributions (zips, tars etc). When an information on a SCM repository is available, the repository creator first tries to check out contents from the respective SCM system. If errors are encountered or if the SCM check out brings no contents, then the repository creator downloads all the packages, given that the information on links to the packages exist in `project.properties`. After the download, the repository creator explodes the archives inside the `content` folder corresponding to the project. The end result of this process is a local Sourcerer repository based on the storage model and that contains contents fetched from remote open source repositories.

**Repository Manager:** The repository manager tool is responsible for two tasks: (i) library management, and (ii) optimizing the local repository for feature extraction. Under library management, the repository manager creates and maintains a local mirror of all jar files from the Maven2 central repository [2] [13]. It also aggregates all of the jar files from the individual projects into the jars directory, as described above. It then creates an index of all the unique jar files in the repository. These jars can be used to provide missing types to projects in Sourcerer's file repository during feature extraction if needed. Under optimizing the local repository, the repository manager performs tasks such as compressing the contents inside a project's folder, and cleaning the jars' manifest files to avoid problems due to unexpected classpath additions.

**Feature Extractor:** The Feature Extractor in Sourcerer is responsible for extracting the detailed structural information from the source code files stored in Sourcerer's file repository. The feature extractor is built as a headless Eclipse plug-in, to make use of Eclipse's AST Parser. Before running the feature extractor, the source code is preprocessed to detect missing libraries using import statements. Some additional heuristics are used to be able to fully resolve the bindings in the source code types and links to the libraries. These heuristics are fully explained in our earlier publication [62]. The Repository Manager and the Feature Extractor together implement the required techniques for Automated Dependency Resolution, a key feature available in the Sourcerer infrastructure, that enables feature extraction from large number of open source projects despite missing dependencies and errors. In summary automated dependency resolution works as follows. First, the feature extraction runs through the available projects to detect missing types. It creates the AST representation of code available in the projects and generates a list of missing types reported by the underlying Eclipse parser. From the list of missing types, the feature extractor generates a list of possible FQNs for those types to be found. It then looks

[2]Maven is a build system for Java that provides the facility to fetch required libraries from a central repository [12].

up the ArtifactDB for possible Jar files where the missing FQNs could be found. While doing so it selects the jar files that can provide the maximum number of missing FQNs. Once the jars are selected, they are included in the classpath of the project with missing types and then the feature extractor runs again. This process is repeated until all missing types are found or if no jars could be located for remaining missing types. After this step, the feature extraction does a full extraction of entities and relations from the projects. Our evaluation of automated dependency resolution has shown that it can increase the percentage of declaratively complete projects in Sourcerer from 39% to 69%. Full details of automated dependency resolution is available in our previous publication [62].

**Database Importer:** This tool allows importing the Feature Extractor's output into the code databases: ArtifactDB and SourcererDB.

**Code Indexer:** The code indexer tool is responsible to index all code entities in Sourcerer's repository using the textual and structural information available for the entities. The code indexer obtains this information using three services, the File Access Service - to obtain the full text corresponding to a code entity, SourcererDB to retrieve entities and comments related to a code entity being indexed, and Similarity Calculation service to retrieve similar entities. As a result of the indexing process, the code indexer produces a semi-structured full text index based on Lucene [6]. Currently the code indexer is implemented as a customization of the Solr [15] indexing and search system. The code index schema varies based on a particular code search application. To index a code entity, the code indexer can retrieve all or some the following data: the full-text for the corresponding entity, the fully qualified names (FQNs) of related entities, comments of the used libraries, and FQNs of used entities. The search index schema will consist of fields to store the terms corresponding to these data types. The terms are extracted from the FQNs and full-text using code-specific analysis techniques (e.g. camel case splitting, removing language keywords as stop words etc). The code indexer tools consists of several of these code-specific analyzers. Appendix B provide further details on configuring the code indexer.

## 6. Applications

The services and stored contents in Sourcerer have been used to develop several code retrieval systems and data mining techniques on source code. The data collected in Sourcerer's file repository have enabled large-scale inter-project analysis techniques that have helped strengthen the capability of the infrastructure itself. This section reviews Sourcerer's key features that facilitated its application in the area of large scale code search, analysis and data mining on source code.

Table 5 lists 6 applications of Sourcerer that have produced major research contributions. The applications are listed in a chronological order. Major publications corresponding to each application is listed in the fourth column. These references provide full details on the specific contributions these applications of Sourcerer has made in the area of Internet-scale code retrieval and source code data mining.

The first column in Table 5 denotes the Sourcerer milestone, indicating the version of the Sourcerer's infrastructure used in these applications. The difference in Sourcerer's infrastructure between milestones M1 and M2 is explained at the end of this section.

The last column in Table 5 indicates whether the corresponding Sourcerer application is available online, and provides the reference to the URL for those that are available online.

*6.1. Impact*

Among the 6 applications in Table 5 Sourcerer Code Search (SCS) Engine, CodeGenie, Structural Semantic Indexing (SSI), and Sourcerer API Search (SAS) fall under the application category of *Internet-scale code retrieval*. The work done in Topic Modeling Source Code is an example of *source code data mining* applications that Sourcerer enables. Finally, Sourcerer Reference Collection (SRC) is an effort towards building *reference collection* for replicable research in large scale code analysis. Together, these applications demonstrate Sourcerer's impact and contribution since its inception.

*6.2. Applications Enabled by Infrastructure Features*

**Reference Collection:** Sourcerer's file repository, storage model, and a simple attribute based project metadata format (see Appendix A) provides a basis for creating a large code repository that can store and describe contents fetched from thousands of open source projects. This enabled us to create and release a recent Sourcerer file repository as a reference collection of source code for research in large-scale analysis of source code. The reference repository is available at [54].

**Data Mining:** The storage model, file repository, relational model, and relational access to SourcererDB allows creating a text-based corpus of source code documents. Each document can have terms extracted from code entities, along with terms extracted from entities that they depend on. This allows rapid construction of a corpus with the bag-of-words representation of documents. The topic modeling work was enabled by the capability of Sourcerer to create many variations of such corpora with little effort.

**Code Indexing:** The index model, repository access, and the relational query service facilitate rapid construction of a search index. The code indexer, being based on Solr, allows building the search index in a declarative way. Two XML files are needed that specify the exact search index (index schema) fields, and data sources to be used to produce a search index (data sources). A wide range of code-specific text analyzers, implemented as part of the Sourcerer infrastructure, can be specified in the index schema to generate terms required for the index fields. Appendix C provides an example of Sourcerer's declarative code indexing.

**Code Retrieval/Search:** All of the five Sourcerer's services enable many required features for building code search applications. First, the code search service can process any standard Lucene query to retrieve a set of code entities from the search index. Again, using Solr as the search server facilitates many advanced search techniques. The format of the search result can be customized as per application's need. Usually the search result contains a subset of matched entities called 'hits', each representing a code entity. Each hit is associated an entity id, which can be used to look up information with the other services. This service-based design facilitates building code search tools that are (i) deeply integrated with developers' working environment, and (ii) leverage textual/structural information extracted from a large amount of source code. Codegenie is an excellent example. Codegenie uses the code search service to first search the entities. It then, using the repository access service, fetches the desired code for each entity returned as a search result. Finally, it gets compilable slices of code using the dependency slicing service that can be merged back into a developer's project workspace. Full details on how Codegenie is built on top of Sourcerer's services is given in [46].

**Inter-project Structural Analysis:** The collection of large number of source code for projects, libraries and automated dependency resolution technique have enabled large scale structural/dependency analysis across projects. The first example of this application is the computation of

global Coderank [19, 18, 48], a measure of popularity for a code entity based on adaptation of Google's Pagerank [43] algorithm on the code-graph created using SourcererDB. In this code-graph, entities represent nodes and relations represent edges. Cross-project links are made by finding the source code implementation for libraries that projects refer to. For example, when a code entity has a relation terminating to a binary (library) entity $l$, the corresponding source entity $l_s$ can be found in the SourcererDB. This allows us to create a global graph of program dependencies that span projects. The details on creating such cross-project links are available in our previous publication [61].

### *6.3. Sourcerer Milestones*

Both the design and implementation of Sourcerer has been constantly evolving. Because of this, earlier applications of Sourcerer use slightly different versions of the models and the repository. At large, the changes in Sourcerer can be divided into two milestones M1 and M2, based on five key features. Table 6 shows these features. Sourcerer M1 existed from 2006 - 2009, and the relational model only captured the entities and relations in Java as defined in Java Language Specification 1.3. These entities and relations were extracted using a custom Java parser built using the JFlex/CUP software packages. Search applications built with Sourcerer M1 used the schema shown in Appendix B, Table C.7. The repository used with Sourcerer M1 had 4632 projects.

Sourcerer M2 is a major rewrite of the entire Sourcerer infrastructure. The original design was kept intact, but the implementation was different, producing a set of loosely coupled tools (described earlier Section 5) unlike a monolithic tool that implemented the functionalities of all current tools in Sourcerer M1. Sourcerer M2's relational model captures the entities and relations as defined Java 1.5. Sourcerer M2 also contains a new and bigger source repository with around 18,000 projects. Sourcerer M2, has been open sourced since 2009. The work on code retrieval done with Sourcerer M2 uses a different code index schema that is described in [21].

## 7. Availability and Access

The implementation of the Sourcerer infrastructure is available online as an open source project [9]. Some of the services are available online as listed in Table 5. As mentioned earlier, we have recently released Sourcerer M2's file repository as a reference collection to be used for research in large scale analysis of source code [54]. Currently, researchers from four different universities have downloaded and are using the repository. Appendix D describes a workflow of using Sourcerer; starting from crawling code to running evaluations using tools available in its implementation. We believe this will motivate other researchers to use and extend Sourcerer in their research.

## 8. Related Work

The work done in code analysis and relational model for source code in Sourcerer is quite similar to the approach taken by many reverse engineering tools and models. For example, FAMIX is a language independent model for describing the static structure of object-oriented software systems [29], and is conceptually compatible with the relational model we use. FAMIXs primary purpose is to support the exchange of information between multiple tools. Where as

| Milestone | Application | Key Idea/Contribution | Publications | Web |
|---|---|---|---|---|
| M1 | Sourcerer Code Search Engine | Coderank; Including Structural Information in code retrieval | [19, 18, 48] | Yes [17] |
| | Topic Modeling Source Code | Extracting concepts and author-topic association | [49, 50, 51, 48] | |
| | | Information-theoretic model of Aspects | [23] | |
| | CodeGenie | Test-driven code search<br>Dependency Slicing | [46, 44, 45, 47] | Yes [8] |
| M2 | Inter-project Structural Analysis | Cross-project linking for determining global usage statistics | [61] | |
| | Structural Semantic Indexing | Effective retrieval of API usage examples leveraging usage similarity | [21] | |
| | Sourcerer API Search | Prototype for Exploratory Code Search using SSI | [22] | Yes [1] |
| | Sourcerer Reference Collection | Reference Collection for research in large scale analysis of source code | | Yes [54] |

Table 5: Various Applications of Sourcerer Infrastructure

| Milestone | Period | JLS | Index Model | Repo. Size | Feature Extractor | Open Sourced |
|---|---|---|---|---|---|---|
| M1 | 2006 - 2009 | 1.3 | SCS | 4K | Homegrown | No |
| M2 | 2009 - 2010 | 1.5 | SSI | 18K, ~350 | Eclipse-based | Yes |

Table 6: Sourcerer Milestones

Sourcerer's relational model's purpose is to represent the structural information in code at the right level of granularity and scale.

Linton's OMEGA system was one of the first to model source code in a relational manner, when back in 1984 he used a relational schema to describe a Pascal-like language [53, 52]. As discussed earlier, Sourcerer's relational model is most closely related to that of Chen et al. [27]. It is also nearly identical to the one used by the back-end repository for [37].

Sourcerer's approach to large scale collection of open source code is, in many ways, similar to Spars-J, a software component repository created by Inoue et al. [42]. Spars-J contains structure and reference information similar to SourcererDB, with the addition of various software metrics. Spars-J also merges similar components, except at the entity, rather than project, level. However the Spars-J public demo appears to be limited to pre-Java 1.5. Although their web-interface allows for searching and browsing of individual files and packages, as well as for references to be followed, it provides no support for direct database or web-service access, making it unsuitable as a foundation to build applications. In addition to Spars-J, there are many other component repositories and code search engines. Merobase, for example, is a commercial component repository that provides a developer API to access its structure-based search [14]. Though it lacks any reference-based information, this still puts it ahead of many other code search engines, such as Google Code Search [11] and Koders [67].

## 9. Future Work

This section presents existing software engineering tools from a few different areas, and describes how they could have benefited from the Sourcerer infrastructure. We also discuss important areas of improvement that could be made to the Sourcerer infrastructure itself.

### *9.1. Possible Benefits to Existing Tools*

A natural extension to Sourcerer's applications would be to reimplement or interface existing software engineering tools with Sourcerer's repository and services. This can not only ease and improve the development of these tools (or future tools like these) but also provide opportunities for fair and more scientific evaluation/comparison of these tools with a common underlying repository.

**Example Recommendation**: Holmes et al.'s Strathcona [37] is a tool for using a developer's current structural context to recommend source code examples. Strathcona attempts to match the structural information in the current context against examples from its repository. The information stored in Strathcona's repository is sufficiently similar to that in Sourcerer's that Strathcona could be implemented on top of the Sourcerer infrastructure. This would focus Strathcona's development on the matching heuristics and client integration, while immediately providing access to a very large repository.

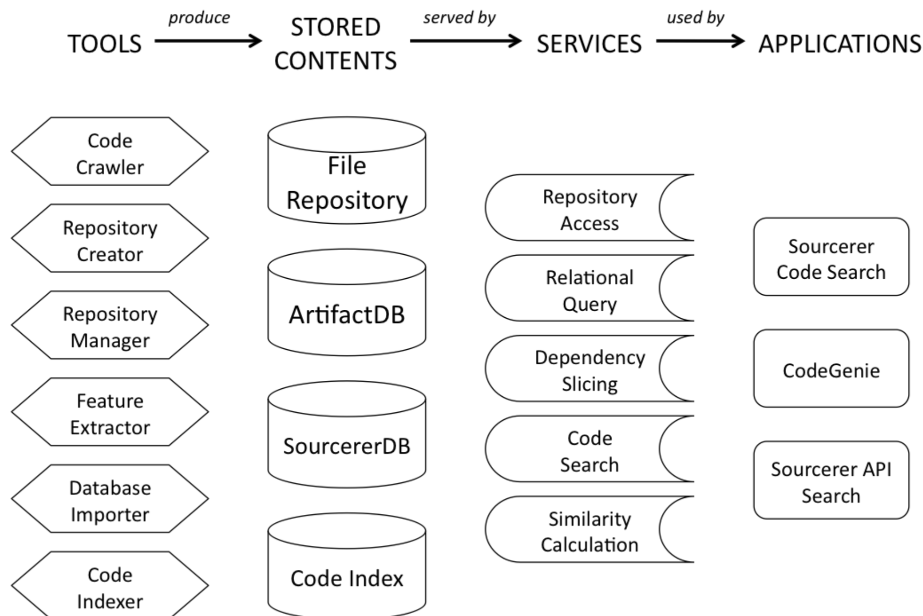



Figure 2: Sourcerer's Overall Architecture

XSnippet [63], Prospector [57] and PARSEWeb [64] are all systems designed to provide examples of object instantiation. Although implementation on top of Sourcerer would provide some benefit to all of them, PARSEWeb would be dramatically improved. Currently PARSEWeb uses Google Code Search to find and download likely examples of object instantiation. These snippets are then analyzed to determine if they contain appropriate invocation sequences. This analysis is complicated by the fact the code snippets are missing most of their external references. PARSEWeb is forced to utilize a variety of heuristic techniques to guess the missing types. Sourcerer is ideally suited for this sort of use, as it can provide snippets where the external references are present, eliminating the errors introduced by the fuzzy analysis.

**Information Mining**: Both SpotWeb [65] and CodeWeb [60] are tools for detecting API hotspots. If they were to use Sourcerer, hotspots could be detected directly simply by ordering the entities in a jar by the number of incoming relations.

**Pragmatic Reuse**: Holmes and Walker's approach to reuse [38] shares many similarities with our dependency slicing. While our approach is fully automated, drawing in all necessary dependencies, theirs permits a greater level of customization, allowing developers to exclude dependencies they do not want. In order to achieve this customization, however, a developer must download and import the full project into his workspace. This creates a fair amount of manual overhead, for if there are multiple candidate projects for reuse, the process must be repeated for each one. Furthermore, any unresolved dependencies in the initial project download will remain unresolved in the final result. The combination of their approach with the Sourcerer infrastructure has the potential to eliminate many of these problems. One could construct a reuse plan on a slice returned by our system, further reducing its size, without having to worry about downloading the full project or unrelated or unresolved dependencies.

*9.2. Extending Sourcerer*

Another area of future work in Sourcerer is to address some of its current limitations, and find opportunities to integrate it with complementary platforms/infrastructure.

**Multiple Languages**: Currently Sourcerer is designed explicitly to work with the Java programming language. Although Java is quite popular as a language of choice to develop software applications today, there are other languages that are equally popular. Even in a single project, it is common to use multiple languages today. For example, a Java project itself might include scripts written in dynamic languages such as Ptyhon and Groovy. Projects also include other declarative mechanisms to generate and build code that make extensive use of languages such as XML. Therefore, models and tools to analyze and store multiple languages, and more importantly, dependencies between them seem to be an important area that an infrastructure such as Sourcerer needs to improve on. In this regard, existing approaches such as XIRC [31] that allows some form of cross artifact analysis in development environments seem relevant. One strategy could be to apply techniques used in XIRC and scale it up to make it work with Sourcerer.

**Addressing Evolution**: The current design of the Sourcerer infrastructure does not have a strong support to address the evolution of the source code. Source code in open source repositories are constantly modified and updated, at least in active projects. Sourcerer's current design requires it to create a separate project for each unique version of project it needs to store. In terms of storage and analysis this is not the most efficient approach to deal with evolving software. A careful survey of the requirements and application of evolutionary data on source code can guide Sourcerer's future approach in dealing with evolution. One possibility would be to extend Sourcerer's current models with elements that describe evolution, as seen in the meta-model used by the Small Project Observatory project [56].

**Considering non-code artifacts**: Source code is not the only artifact that is available in open source repositories. It is well known that during a software development lifecycle various kinds of non-source artifacts are produced and used. For example, data on issues, bugs, documentation, authorship, developer's activities/ history etc. While Sourcerer is not primarily designed to address these kinds of non-source artifacts, it is important to find an approach to connect Sourcerer's models and services with repositories (for example Hipikat [28] that store these non-code artifacts.

**Integrating with other open source analysis platforms**: Another avenue of extension would be to integrate Sourcerer with open-source quality monitoring platforms, such as Alitheia Core [33], that aim to collect source code metrics from open source projects on the Internet. Such integration could enable yet new applications, that would leverage open-source quality metrics with information available in Sourcerer's stored contents.

The FLOSSmole project is a collaborative effort to collect and analyze large amount of open source project data [39]. Sourcerer is more comprehensive in terms of tools, services and depth of information it covers for analyzing the source code available in open source project. FLOSSmole's database include more project specific metadata, and had information on larger number of open source projects. On one hand, the data from FLOSSmole project could be used, instead of the data produced by Sourcerer's code crawler, to construct a Sourcerer file repository. On other hand, the depth of information that can be produced using Sourcerer can further enhance the kinds of analyses that FLOSSmole currently supports. Therefore, integrating Sourcerer with FLOSSmole could widen the scope and impact of both projects.

## 10. Conclusion

This paper presented the design and implementation of the Sourcerer infrastructure. The primary components and the basic working principle of Sourcerer can be summarized in the architecture diagram presented in Figure 2. Sourcerer's architecture consists of a set of loosely coupled tools that produce various stored contents. These stored contents confirm to the design specified by three models: Storage, Relational and Index. Five different services provide a layer of abstraction for programmatic access to the underlying stored content. These services enable the development of applications that require access to preprocessed information from large amount of source code (in both textual and structural form).

The Sourcerer infrastructure makes three primary contributions in the area of large-scale collection, analysis and application of open source code:

1. **Collection**: The storage model, a common metadata format for describing open source projects across various online repositories, and the plugin-based design of the code crawler enable collection of large-amount of code from the Internet.
2. **Analysis**: A large collection of libraries in Sourcerer's file repository, and the automated dependency resolution implemented in the feature extractor increases the number of declaratively complete projects to 69% from 39% [62]. This makes it possible to extract fine-grained structural information from a large number of projects even in the presence of missing dependencies.
3. **Applications**: Sourcerer consists of several services that provide a layer of abstraction and programmatic access to textual, structural, and information retrieval models of source code. This enables rapid development and evaluation of novel techniques for source code retrieval and data mining. By providing the required models, tools, and services - it serves as a testbed for rapid implementation and evaluation of large scale code analysis tools such as Internet-scale code search tools. The successful application of Sourcerer in applications such as Sourcerer Code Search engine [19, 18, 48] , Codegenie [46, 44, 45, 47], and Structural Semantic Indexing validates [21] Sourcerer's impact in this area.

Besides above three contributions, Sourcerer provides key resources to conduct replicable research in large scale code analysis by (i) making available a large collection of source code in a standard format [54], and (ii) by making the infrastructure's implementation available as an open source project [9]. Finally, we hope that the details on the Sourcerer's implementation available in the Appendix will motivate external usage and contribution to the Sourcerer project.

**Acknowledgements:** Sourcerer is a large systems project, incorporating over 8 people (up to 4 at a given time). The first two authors are responsible for all work done in Sourcerer since milestone M2. We acknowledge contributions made by other in the earlier versions of Sourcerer infrastructure.

*Contributors to Sourcerer milestone M1:* Yimeng Dou implemented the first basic version of the Sourcerers crawler that was replaced by a more robust and plugin-based code crawler by Huy Huynh. Trung Ngo was instrumental in contributing to the first design and implementation of the overall Sourcerer infrastructure and developed the context sensitive analysis, core indexing components, and the 'Coderank' [18, 19, 48] technique that was part of the graph-based heuristic to improve code retrieval in SCS. Paul Rigor contributed to deployment and running our tools on various system configurations ranging from a single machine to clusters of machines. Erik Linstead was responsible for running the experiments required for evaluation of retrieval schemes

in SCS [19, 48], and was the primary contributor to data-mining applications of Sourcerer [49, 50, 51, 48, 23]. Otavio Lemos, contributed to the idea of of Test-driven Code Search (TDCS) and implemented CodeGenie [46, 44, 47, 45]. Last but not the least, Pierre Baldi led the data-mining research in Sourcerer and supported the infrastructure's implementation since its inception.

## References


[1] Sourcerer wiki page on sourcerer api search tool *http://wiki.github.com/sourcerer/Sourcerer/sas*.

[2] Web Site for Apache Software Foundation. *http://apache.org*.

[3] Web site for Java.net. *http://java.net*.

[4] Web site for Tigris. *http://tgris.org*.

[5] Black Duck's web page with Koders usage information. http://corp.koders.com/about/, February 2010.

[6] Lucene web site. *http://lucene.apache.org*, Jan 2010.

[7] Web Location of Galago Search Evaluation Tool. *http://code.google.com/p/galagosearch/source/browse/tags/galagosearch-1.04/galagosearch-core/src/main/java/org/galagosearch/core/eval/Main.java*, July 2010.

[8] Web page for codegenie. *http://sourcerer.ics.uci.edu/codegenie*, July 2010.

[9] Web page for Sourcerer's github repository. *http://github.com/sourcerer/Sourcerer*, June 2010.

[10] Web Page on Apache Lucene Scoring. *http://lucene.apache.org/java/2_4_0/scoring.html*, Mar 2010.

[11] Web site for Google Code Search. *http://www.google.com/codesearch*, July 2010.

[12] Web site for maven. *http://maven.apache.org*, July 2010.

[13] Web site for maven's central repository. *http://repo1.maven.org/maven2/*, July 2010.

[14] Web site for merobase. *http://www.merobase.com/*, July 2010.

[15] Web Site for Solr *http://lucene.apache.org/solr/*, July 2010.

[16] Web site for Sun Grid Engine. *http://gridengine.sunsource.net*, July 2010.

[17] Web page for sourcerer project and the sourcerer code search engine. *http://sourcerer.ics.uci.edu*, July.

[18] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. pages 681–682, New York, NY, USA, 2006. ACM Press.

[19] S. Bajracharya, T. Ngo, E. Linstead, P. Rigor, Y. Dou, P. Baldi, and C. Lopes. A study of ranking schemes in internet-scale code search. Technical Report UCI-ISR-07-8, UCI ISR, November 2007.

[20] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4. IEEE Computer Society, 2009.

[21] S. Bajracharya, J. Ossher, and C. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. 18th International Symposium on the Foundations of Software Engineering, 2010.

[22] S. Bajracharya, J. Ossher, and C. Lopes. Searching api usage examples in code repositories with sourcerer api search. In *SUITE 2010: Second International Workshop on Search-driven Development - Users, Infrastructure, Tools and Evaluation*, 2010.

[23] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 543–562, Nashville, TN, USA, 2008. ACM.

[24] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th international conference on Human factors in computing systems*, pages 1589–1598, Boston, MA, USA, 2009. ACM.

[25] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceegings of FSE*, pages 213–222, Amsterdam, The Netherlands, 2009. ACM.

[26] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. 2009.

[27] Y. Chen, E. R. Gansner, and E. Koutsofios. A c++ data model supporting reachability analysis and dead code detection. *IEEE Trans. Softw. Eng.*, 24(9):682–694, 1998.

[28] D. Cubranic, G. Murphy, J. Singer, and K. Booth. Hipikat: a project memory for software development. *Software Engineering, IEEE Transactions on*, 31(6):446–465, 2005.

[29] S. Demeyer, S. Tichelaar, and S. Ducasse. Famix 2.1 — the famoos information exchange model. Technical report, University of Bern, 2001.

[30] A. Deshpande and D. Riehle. The total growth of open source. In *Fourth Conference on Open Source Systems*. Springer Verlag, 2008.

[31] M. Eichberg, M. Mezini, K. Ostermann, and T. Schafer. Xirc: A kernel for cross-artifact information engineering

in software development environments. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 182–191, Washington, DC, USA, 2004. IEEE Computer Society.

[32] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The*. Addison Wesley, 3 edition, June 2005.

[33] G. Gousios and D. Spinellis. Alitheia Core: An extensible software quality monitoring platform. In *Proceedings of the 31st International Conference on Software Engineering*, pages 579–582. IEEE Computer Society, 2009.

[34] M. Grechanik, K. M. Conroy, and K. A. Probst. Finding Relevant Applications for Prototyping. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 12. IEEE Computer Society, 2007.

[35] J. Hammond. What developers think. http://www.drdobbs.com/architect/222301141, Jan 2010.

[36] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 13–22, Newport, Rhode Island, USA, 2007. ACM.

[37] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM Press.

[38] R. Holmes and R. J. Walker. Lightweight, Semi-automated Enactment of Pragmatic-Reuse Plans. In *Proceedings of the 10th international conference on Software Reuse: High Confidence Software Reuse in Large Systems*, pages 330–342, Beijing, China, 2008. Springer-Verlag.

[39] J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[40] http://sourcerer.ics.uci.edu/services. Sourcerer web services.

[41] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Softw.*, 25(5):45–52, 2008.

[42] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.

[43] R. M. Lawrence Page, Sergey Brin and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Stanford Digital Library working paper SIDL-WP-1999-0120 of 11/11/1999 (see: http://dbpubs.stanford.edu/pub/1999-66)*.

[44] O. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, 2009.

[45] O. A. L. Lemos, S. K. Bajracharya, and J. Ossher. CodeGenie:: a tool for test-driven source code search. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 917–918, Montreal, Quebec, Canada, 2007. ACM.

[46] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, and C. V. Lopes. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology*, (To appear).

[47] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. CodeGenie: using test-cases to search and reuse source code. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, Atlanta, Georgia, USA, 2007. ACM.

[48] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, Apr. 2009.

[49] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 461–464, Atlanta, Georgia, USA, 2007. ACM.

[50] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining eclipse developer contributions via Author-Topic models. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 30. IEEE Computer Society, 2007.

[51] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining internet-scale software repositories. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 929–936, Cambridge, MA, 2008. MIT Press.

[52] M. A. Linton. Queries and views of programs using a relational database system. Technical Report UCB/CSD-83-164, EECS Department, University of California, Berkeley, 1983.

[53] M. A. Linton. Implementing relational views of programs. In *SIGPLAN Not.*, page 132–140, New York, NY, USA, 1984. ACM Press.

[54] C. V. Lopes, S. K. Bajracharya, J. Ossher, and P. F. Baldi. UCI Source Code Data Sets. [http://www.ics.uci.edu/~lopes/datasets/] Irvine, CA: University of California, Bren School of Information and Computer Sciences.

[55] Lucid Imagination. Lucidworks for solr certified distribution reference guide. http://www.lucidimagination.com/Downloads/LucidWorks-for-Solr/Reference-Guide, 2010.

[56] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264–275, Apr. 2010.
[57] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61, New York, NY, USA, 2005. ACM Press.
[58] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 1 edition, July 2008.
[59] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action*. Manning Publications, 2 edition, July 2010.
[60] A. Michail. Code web: data mining library reuse patterns. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 827–828, Toronto, Ontario, Canada, 2001. IEEE Computer Society.
[61] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. SourcererDB: an aggregated repository of statically analyzed and cross-linked open source java projects. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 183–186, 2009.
[62] J. Ossher, S. Bajracharya, and C. Lopes. Automated dependency resolution for open source software. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 130–140, Cape Town, South Africa, 2010.
[63] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 413–430, New York, NY, USA, 2006. ACM Press.
[64] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 204–213, Atlanta, Georgia, USA, 2007. ACM.
[65] S. Thummalapenta and T. Xie. SpotWeb: detecting framework hotspots and coldspots via mining open source code on the web. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 327–336, 2008.
[66] Web Site for Google Code Hosting. *http://code.google.com/projecthosting*.
[67] Web site for Koders. *http://www.koders.com*, 2010.
[68] Web site for Krugle. *http://www.krugle.com*, 2010.
[69] Web Site for Sourceforge. *http://sourceforge.net*.

## Appendix A. Sample project.properties files

The example below shows metadata description for a project crawled from Google code hosting.

```
00 #Thu Sep 24 16:15:01 PDT 2009
01 releaseDate=null
02 name=dlctarea1
03 category=DLC, Java, Netbeans, FileChooser
04 languageGuessed=Java
05 versionGuessed=$SCM
06 scmUrl=svn checkout
    http\://dlctarea1.googlecode.com/svn/trunk/
      dlctarea1-read-only
07 license=GNU General Public License v2
08 keywords=null
09 sourceUrl=null
10 exractedVersion=$SCM
11 projectDescription=Tarea n\uFFFD 1
12 fileExtensions=null
13 originRepositoryUrl=http\://code.google.com
14 containerUrl=http\://code.google.com/p/dlctarea1/
15 contentDescription=null
16 crawledDate=2009-Sep-23
```

The example below shows metadata description for a project crawled from Sourceforge. This description includes list of downloadable packages along with the SCM URL, unlike above that only has the SCM URL.

```
00 #Tue Sep 22 23:53:26 PDT 2009
01 versionGuessed=$SCM
02 containerUrl=http\://sourceforge.net/cvs/?group_id\=25954
03 projectDescription=\  The Virtual Data Center project is
    building an operational, open-source, digital library to
    enable the sharing of quantitative research data, and the
    development of distributed virtual collections of data
    and documentation.
04 contentDescription=cvs
05 package.releaseDate.2=2006-04-17 20\:23
06 package.releaseDate.1=2006-04-17 20\:23
07 license=GNU General Public License GPL
08 languageGuessed=Java , Perl
09 keywords=null
10 exractedVersion=$SCM
11 package.sourceUrl.2=
    http\://downloads.sourceforge.net/thedata/
      VDC-1.0.4-11.Fedora.4.tar
12 crawledDate=2009-Feb-28
13 package.sourceUrl.1=
    http\://downloads.sourceforge.net/thedata/
      VDC-1.0.4-11.RedHat.4AS.tar
14 package.versionGuessed.2=VDC 1.0.4 Final
15 package.versionGuessed.1=VDC 1.0.4 Final
16 fileExtensions=null
17 category=Education , Dynamic Content , Indexing/Search ,
    Other/Nonlisted Topic , Scientific/Engineering , Archiving
18 package.name.2=thedata-vdcfedora
19 package.extractedVersion.2=VDC 1.0.4 Final
20 package.name.1=thedata-vdcredhatas
21 package.extractedVersion.1=VDC 1.0.4 Final
22 sourceUrl=null
23 releaseDate=null
24 originRepositoryUrl=http\://sourceforge.net
25 scmUrl=
     cvs -d\:pserver\:anonymous@thedata.cvs.sourceforge.net\
       :/cvsroot/thedata login;
     cvs -z3 -d\:pserver\:anonymous@thedata.cvs.sourceforge.net\
       :/cvsroot/thedata co -P modulename
26 package.size=2
27 name=thedata
```

These two examples show that Sourcerer's project metadata format enables description of projects and contents across various online repositories.

## Appendix B. Sourcerer Code Index Schemas

Table C.7 shows Sourcerer's code index model used in Sourcerer Code Search and CodeGenie.

## Appendix C. Declarative Code Indexing with Solr

This section shows how Sourcerer provides a declarative way of defining and creating a code index using Solr. The syntax, specification and declarative approach is due to the use of Solr.

Three kinds of specifications are required for creating a search index in Sourcerer. First, every index field needs to define a field type.

**Field Declaration:** The XML fragment below shows a field declaration named `sname_contents` that has a field type of `FT_sname_contents`. The field stores the terms extracted from a simple name of code entity, the text source from which the field is populated is a FQN of a code entity. This is retrieved from SourcererDB, details are shown later in this section.

```
<field name="sname_contents" type="FT_sname_contents"/>
```

**Field Types:** A number of fields can be declared once a field type is defined. A field type specifies the analysis done on the textual content that will be stored in the corresponding field. For example, the field type `FT_sname_contents` defined below denote the field type to store the terms extracted from the simple name of Java code entities. These terms are extracted from the FQNs of entities that are retrieved from SourcererDB.

```
01 <fieldType name="FT_sname_contents"
02       class="solr.TextField">
03   <analyzer type="index">
04     <tokenizer
05       class="solr.KeywordTokenizerFactory" />
06     <filter class="sourcerer.FqnFilterFactory"
07       extractSig="0" shortNamesOnly="1" />
08     <filter class=
09      "sourcerer.NonAlphaNumTokenizerFilterFactory"/>
10     <filter class=
11       "sourcerer.CamelCaseSplitFilterFactory"/>
12     <filter class=
13       "sourcerer.LetterDigitSplitFilterFactory"
14       preserveOriginal="1" />
15     <filter class="solr.LowerCaseFilterFactory"/>
16   </analyzer>
17   <analyzer type="query">
```

The field type definition shown above indicates that a FQN (an input text to the field) is treated as a Keyword token (due to the use of the class shown in Line 05). Therefore, the analysis starts with the full FQN in its original form. The FQN is then piped through five filters. Each filter takes a token stream (a stream of terms), processes it and produces a new token stream. The terms produced at the end is what gets stored in a field. The values of the attribute ‘class’ in the XML fragment above (Lines 06 - 15) denote a Java implementation of a token filter to use. Classes with the prefix ‘sourcerer’ are code specific token filters available as part of Sourcerer’s implementation. In the above example, the first filter (lines 06 and 07) extracts the simple name from the FQN, a second filter (Line 08) splits the simple name using non-alphanumeric characters as delimiters, the third filter does a camel case split on the new token stream, the fourth filter split the term further based on letter-digit transition while keeping the terms already present in the token stream, and the last filter converts every term to a lowercase version. The output from this last filter gets stored as terms corresponding to a simple name of a Java entity.

**Data Source Configuration:** Another step in declarative specification for indexing is the configuration of data sources that provide the content from which terms will be extracted and stored in the index fields. The XML snippet below shows how SourcererDB and the repository access service is used to feed data into some index fields.

```
01 <document name="entity">
02   <entity name="code_entity" pk="entity_id"
03     query="SELECT entity_id, fqn FROM entities
04       WHERE entity_type in
05           ('CLASS','METHOD','CONSTRUCTOR')''
06     transformer="sourcerer.Transformer"
07     code-server-url=
08       "http://sourcerer-url/file-server">
09   <field column="fqn" name="sname_contents" />
```

```
10   <field column="fqn" name="fqn"/>
11   <field column="entity_id" name="entity_id" />
12   <field column="code_text" name="full_text"/>
```

Above, Line 02 indicates that a document in the index represents a code entity. Line 03 specifies a SQL query that will fetch a set of columns from SourcererDB. Lines 09 - 12 indicate the number of fields that will be stored for the corresponding document. The values for the attribute 'column' specifies the column name corresponding to the SQL query that start in Line 03. The values for the attribute 'name' indicates the index field the contents from the column will be stored into. Line 12 indicates that the column to be used to get the content for the index field `full_text` is named 'code_text' that does not exist in the SQL query starting at Line 03. This column is added by the class named `sourcerer.Transformer`, which acts as a data transformer. The class is responsible to fetch the source code corresponding to an entity. It uses the value from the column `entity_id`, sends an HTTP request to the repository access service available at the URL specified in Line 07, and provides the text as an input to be stored for index field `full_text`.

This demonstrates how easily a searchable index can be created once the services are accessible in Sourcerer. The XML snippets shown above are simplified versions of what exists in Sourcerer's indexing tool implementation. The first two snippets come from the schema definition file (named schema.xml in a Solr installation), the third snippet comes from the data import file (named db-data-config.xml in a Solr installation). More details on configuring these files are available from standard references on Solr [55, 15].

## Appendix D. Example Sourcerer Workflow

This section presents how various parts of Sourcerer's implementation (available in Github at http://github.com/sourcerer/Sourcerer) can be used in evaluation of a code retrieval scheme. It starts from running the code crawler to build a file repository to running a client application. The workflow resembles the setup/scenario of evaluation done for Structural Semantic Indexing as described in [21].[3]

**Creating a Sourcerer File Repository:** The two projects found under *infrastructure/tools/core* are needed to create the Sourcerer file repository. The projects *codecrawler* and *core-repository-manager* are used to crawl and build the file repository. *codecrawler* implements the code crawler tool, and *core-repository-manager* implements the Repository Creator tool. Binaries to run these tools are found in the Github repository folder *bin*.

Once a file repository is in place, the following projects are needed to perform automated dependency resolution: *repository-manager* - that implements the Repository Manager tool, *extractor* - that implements the Feature Extractor, *model* - that implements the Relational Model, *database* - that implements the Database Importer tool, and *utilities*. These projects can all be found under *infrastructure/tools/java*, except for *utilities*, which is found under *infrastructure*. In order to use a Sourcerer file repository created using the Repository Creator Tool, some preprocessing of the repository is necessary. The jar files from the projects must be aggregated and then indexed for quick access. This is done by running `edu.uci.ics.sourcerer.repo.Main` twice,

[3] The repository used in [21] was partly created manually. The files were added to a folder structure confirming Sourcerer's storage model.

| Index Field | Description |
|---|---|
| *Fields for basic retrieval* | |
| contents | Default field to be searched. combination of fqn_contents, fqn_fragments and fqn |
| fqn | Fully qualified name of an entity, untokenized |
| fqn_contents | Tokenized terms from the FQN of an entity. |
| fqn_fragments | untokenized form of FQN fragments. Eg: for a FQN; foo.bar.SomeOne.method(int) - the fragments are; foo bar SomeOne method |
| short_name | right most fragment of the FQN (w/o method arguments for methods) |
| short_name_contents | tokenized form of short_name |
| comments | The collected text (untokenized) from an entity's comments |
| *Fields for retrieval with signatures* | |
| m_sig_args_fqn | method's formal arguments FQN in format org.foo.Arg1,x.y.arg2,z.arg3,...,Y.argn |
| m_sig_args_sname | method's formal arguments short name in format arg1,arg2,arg3,...,argn |
| m_sig_ret_type_sname | short name of the method's return type, rightmost of FQN |
| m_sig_ret_type_fqn | FQN of the method's return type |
| m_args_arity | Number of arguments a method has |
| m_args_fqn_fragments | method's arguments' FQN fragments (not ordering info) |
| m_args_fqn_contents | method's formal arguments tokenized into terms after passing each of them through keyword extractor |
| m_args_sname_contents | Terms extracted from the short names of each method argument |
| m_ret_type_fqn_fragments | FQN fragments of return type |
| m_ret_type_contents | method's return type FQN tokenized into terms after passing it through keyword extractor |
| m_ret_type_sname_contents | short name of the method's return type tokenized |
| *Fields Storing metadata* | |
| modifiers | modifiers applied on this entity |
| is_binary | true for entities coming from jars/classes and those that are missing, false for entities coming from source code |
| entity_type | String representation of entity type. Eg; "CLASS" |
| *Fields storing metrics* | |
| complexity | currently mapped to lines of code |
| coderank_local | local coderank |
| *Fields for navigation* | |
| entity_id | the entity id of this entity in the code database |
| parent_id | the parent entity that contains this entity (via the inside relation) |
| fan_in_all_local | all incoming entity ids |
| fan_out_all_local | all outgoing entity ids |
| fan_in_mcall_local | entity ids of all local callers for a method from the same project |
| fan_out_mcall_local | entity ids of all local callees for a method from the same project |

Table C.7: Fields in Sourcerer's code index schema used in Sourcerer Code Search and Codegenie

| Step | Task | run *Script*/`Class`/service | found in *folder* / `package` | part of.. |
|---|---|---|---|---|
| 1 | crawl projects | *run-code-crawler.sh* | bin/ | Code Crawler |
| 2 | create repository layout | *repo-folder-creator.sh* | bin/ | Repository Creator |
| 3 | populate repository | *content-fetcher.sh* | bin/ | Repository Creator |
| 4 | prepare repository for extraction | `Main.java` | `edu.uci.ics.sourcerer .repo` | Repository Manager |
| 5 | feature extraction for ArtifactDB | `Extractor.java` | `edu.uci.ics.sourcerer .extractor` | Feature Extractor |
| 6 | populate ArtifactDB | `Main.java` | `edu.uci.ics.sourcerer .db.tools` | Database Importer |
| 7 | run ArtifactDB | MySql Database | | Relational Access |
| 8 | full feature extraction | `Extractor.java` | `edu.uci.ics.sourcerer .extractor` | Feature Extractor |
| 9 | populate SourcererDB | `Main.java` | `edu.uci.ics.sourcerer .db.tools` | Database Importer |
| 10 | run SourcereDB | MySql Database | | Relational Access |
| 11 | get raw usage data for Hamming Distance/Tanimoto Coefficient based similarity | *run-raw-usage-writer.sh* | bin/ | Similarity Calculation |
| 12 | get final usage data for Hamming Distance/Tanimoto Coefficient based similarity | *run-filtered-usage-writer.sh* | bin/ | Similarity Calculation |
| 13 | prepare index for TF-IDF similarity | *run-index.sh* | infrastructure/services/solr-config | Code Indexer |
| 14 | run TF-IDF based similarity service | Solr Instance | infrastructure/services/solr-config | Similarity Calculation |
| 15 | run HD/TC based similarity service | `SimilarityServer.java` | `edu.uci.ics.sourcerer .server.similarity` | Similarity Calculation |
| 16 | run Repository access | `FileServer.java` | `edu.uci.ics.sourcerer .server.file` | Repository Access |
| 17 | run final indexing | *run-index.sh* | infrastructure/services/solr-config | Code Indexer |
| 18 | run code search service | Solr Instance | | Code Search |
| 19 | run retrieval evaluation for SSI | `EvalSnippetsEntryPoint .java` | `edu.uci.ics.sourcerer .evalsnippets.client` | Applications |

Table C.8: An example Sourcerer Workflow

with the aggregate-jar-files and create-jar-index flags respectively. In each case, the input repository (input-repo) must be specified.

**Building ArtifactDB:** As specified by the storage model, the jars folder in the file repository contains the jar files used to resolve the missing dependencies. The contents of this folder and the corresponding Jar-index can be constructed from any managed repository, which can contain whatever artifacts the user wants, such as a mirror of the Maven Central Repository. We recommend contacting the people at Apache in order to obtain such a mirror, though a crawler and downloader for it can be found in the *repository-manager* project in the edu.uci.ics.sourcerer.repo.maven package. Once the file repository is ready with the Jar-index, the Feature Extractor tool is used to extract the types provided by these artifacts. The Feature Extractor is run as an Eclipse application, class Extractor inside package edu.uci.ics.sourcerer.extractor , which is found in the *extractor* project. It can either be used directly in Eclipse, or as a headless plugin. The following must be specified in command-line arguments: the type of extraction (extract-jars to limit to jar files and extract-binary to ignore jar file source), the input file repository (input-repo), and the output repository (output-repo). After the extraction is complete, an instance of ArtifactDB can be populated with the type information. This is done using class Main, inside package edu.uci.ics.sourcerer.db.tools, in the *database* project.

**Automated Dependency Resolution:** Once ArtifactDB is ready, the Feature Extractor tool can perform automated dependency resolution. Dependency resolution is available for both jar and project extraction, simply by adding the resolve-missing-types flag to the Extractor. If dependency resolution is used, the database containing the ArtifactDB must also be specified as an argument to Extractor.

**Building SourcererDB:** Populating SourcererDB is done in exactly the same way as ArtifactDB.

**Running Services:** The File Repository and SoumererDB make up two content sources needed for indexing. To proceed with indexing, first the file repository needs to be served using the Repository Access service. An implementation of this service is available in the Github repository under *infrastructure/ services/ file-server*.

To create the search index required for retrieval evaluation described in [21], two different similarity calculation services are needed. First, that computes the Tanimoto Coefficient/Hamming Distance based similarity. This part is implemented in *infrastructure /services /similarity-server*. The *similarity-server* requires a data source with information on API usage. This can be provided as a plain text file. A tool to generate this usage information is available inside project *infrastructure /tools /core /machine-learning*. Two classes produce the required usage statistics. First, class UsagePreCalculatorRunner inside package edu.uci.ics.sourcerer.ml.db.tools calculates the usage details on all entities and APIs. Second, class FilteredUsageCalculator found in the same package finally prepares the required usage statistics by filtering outliers (e.g.; APIs that are used by only one entity).

The second similarity calculation service that computes the TF-IDF based similarity requires a running instance of a minimal code search service that is available under *infrastructure /services /solr-config*. The procedure required to run the indexing tool and the code search service is same as for running these tools to do the final retrieval evaluation. The only difference lies in writing the schemas and data configuration specification.

**Indexing**: The implementation of the code indexer and code search tools is available under *infrastructure /services /solr-config*. To run the code index tool, first a Solr installation needs to be configured by writing schemas as described Appendix B. The script that runs the search tool is available in the Github project *solr-config*, and is named *runindex.sh*. This script runs the indexing in a cluster of machines running the Sun Grid Engine distributed computing platform [16]. The script can easily be modified to run the indexing in a single machine.

**Retrieval and Evaluation**: The retrieval step requires four different services to be running: Repository Access (*file-server* in Github), Relational Query (an instance of SourcererDB), similarity server for HD/TC similarity (*similarity-server*) in Github, and similarity server for TF-IDF similarity (a configured instance of *solr-config*). Once these services are up and running, an instance of the retrieval evaluation tool can be run to execute different queries and collect the required statistics that get generated. The retrieval tool is available in Github under project *infrastructure /apps /codesearch*. The implementation of the retrieval tool used in [21] is available as a GWT (Google Web Toolkit) based web application starting at class EvalSnippetsEntryPoint in package edu.uci.ics.sourcerer.evalsnippets.client.

The output of running the retrieval evaluation tool can be processed using a tool available in Galago Search project [7]. Finally, the output from Galago Search project provides the necessary data that can be analyzed to measure the retrieval performance. To ease this analysis, handy scripts to be used with the R statistical programming environment are available in Sourcerer's Github location *research /api-location /evaluation*.

**Summary**: The workflow described above demonstrates that Sourcerer provides an end-to-end solution for large scale collection, analysis and retrieval of code. A collection of tools and services are required for that purpose. The implementation of these tools and services are found in Sourcerer's github repository but can be difficult to trace through in the order they need to be used. Therefore, in Table C.8, we present a summary of the workflow described above in 19 steps that need to be performed in the given order. The table shows the name and location of the implementation for scripts, tools and services corresponding to each step. It also shows the part of the Sourcerer's architecture that each

implementation belongs to.

# SyLaGen: From Academic Tool Engineering Requirements to a new Model-based Development Approach

Moritz Balz[a], Michael Striewe[a], Michael Goedicke[a]

[a]*Paluno – The Ruhr Institute for Software Technology*
*University of Duisburg-Essen, Campus Essen, Germany*

---

**Abstract**

In this contribution we reflect on the development of SyLaGen, an academic load generation tool for performance tests. It is able to generate a defined amount of requests to a system under test and measure the response times. The development of this tool has been influenced by two facts over the last ten years: First, its variety in functionality and the high number of platforms and frameworks in use; and second, the desire to specify the main functionality for measurements as precise as possible with respect to appropriate models. However, these requirements often contradict, since model-driven development is not easy to apply to existing architectures. In the case of our tool, this lead to a different approach for model-based development embracing formalized design patterns. We will here introduce the nature of this academic tool and the side effects of its development to other software engineering domains.

---

## 1. Introduction

Tool building in academia can be driven by different factors: functional requirements, non-functional requirements, experimental development processes, case studies, or others. If industrial partners are involved, all of these factors can be influenced not only from inside academia, but also from the outside, including change requests and time constraints. In this contribution we reflect on the development of SyLaGen, which has experienced several development steps during the past ten years, some of them with project partners from the industry. The development has been influenced by functional and non-functional requirements as well as by development styles, which makes it an interesting case study for academic tool building. One of the most important findings was the creation of a new implementation style for model-based programming that will be explained in detail in this paper.

In general terms, SyLaGen (the name is an acronym for the German tool description "Synthetischer Last-Generator") is a load generator application for performance tests. It is able to generate a defined amount of requests to a system under test and measure the response times. Several important capabilities are explained in detail in sections 2.1 and 3.1. More general information can be found in existing publications, one referring to an initial version [3] and one to a re-designed version [11].

*Email addresses:* moritz.balz@uni-due.de (Moritz Balz), michael.striewe@s3.uni-due.de (Michael Striewe), michael.goedicke@s3.uni-due.de (Michael Goedicke)

This contribution is organized as follows: Section 2 reports on the initial development of SyLaGen to give a feeling for the initial requirements and the nature of the load generation tool. Section 3 elaborates on the re-development that took place some years later as a pure academic project. Section 4 provides the main contribution of this paper and focuses on the most important findings on development and implementation style during the re-development phase. Section 5 concludes the paper.

## 2. Initial development of SyLaGen

The initial development of SyLaGen was started as a cooperation between the University of Duisburg-Essen and the Siemens company in the late 1990s. Thus, the first version of SyLaGen had to meet both academic and industrial requirements. In addition, the development process had to obey limited project resources and some engineering and development standards used by Siemens.

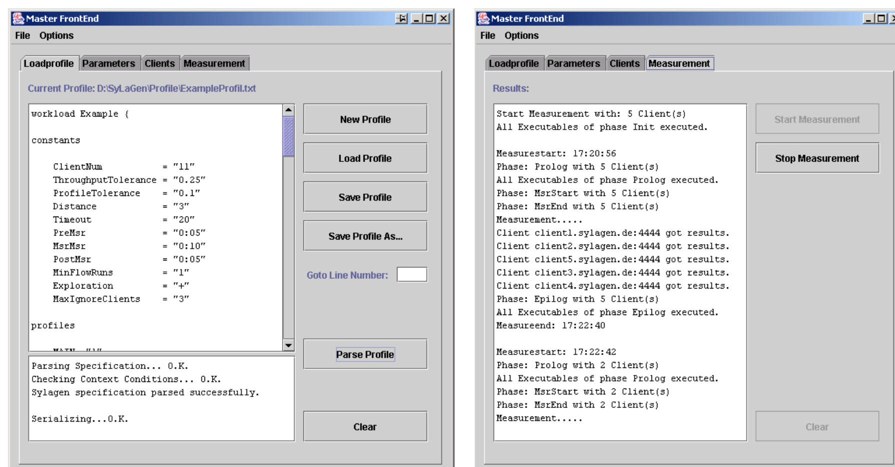
### *2.1. Requirements of* SyLaGen

From its beginning, SyLaGen was intended to be a framework rather than a monolithic tool and thus focused on extendability with respect to four different aspects: First, a system under test may offer different interfaces for handling external requests, thus a load generator must be able to handle different protocols randomly and in parallel. This was the major key argument for developing a new load generator at all, because Siemens wanted to be able to test their proprietary protocols, which was not possible with available tools. Second, load generation for a client-server system may require complex client behaviour that cannot be formulated in a simple descriptive way, but instead with non-trivial algorithms that have to be implemented programmatically. Third, more than simple atomic measurements may be required in complex environments, so that strategies applying sequences of measurements to a system should be configured. In particular, these strategies may incorporate preparations for each single measurement, e.g. resetting databases, or adjust load generation parameters between two atomic measurements. Finally, complex load generation may result in complex use cases that cannot be modelled as linear scripts, but as probabilistic networks.

Some additional properties regarding the general system design were required in parallel to the flexibility requirements named above: To ease generation of high load volumes, the system architecture must allow distributed load generation from different clients. For this purpose a "master" component controlling the measurement should connect to client instances that generate the actual load and may be either physical computers or logical instances sharing a host system.

### *2.2. Design Implementation of the Initial Version*

The implementation of the first version of SyLaGen was straightforward: The master component was a monolithic desktop application based on Java's Swing user interface. The workload was specified textually as shown in figure 1(a). A parser for the workload input format was generated automatically with JavaCC [5] from a grammar description. Besides several parameters like measurement time, timeouts and so on, a workload was composed of so-called "flows". Each flow can be understood as a simple state machine where each state contains load generating requests to the system under test. Transitions are labeled with a weight, allowing for probabilistic paths through these machines. This provides the required flexibility for complex load generation which is far more powerful than linear scripts.




(a) Textual workload definition in the first version of SyLaGen.

(b) Textual measurement output in the first version of SyLaGen.

Figure 1: The user interface of the first SyLaGen version.

Similar to the textual input, during measurement the output was also displayed as a log file on the screen as shown in figure 1(b). The results of the complete measurement were stored in plain text files since sophisticated reporting was outside the scope of the requirements.

The client components were written in C to allow for efficient execution even on limited devices and were connected to the master with a simple socket-based network protocol. In order to achieve the desired flexibility for protocols used in requests to systems under test, so-called "adapters" were used. Each adapter was implemented in a single DLL library and provided a set of methods to access a certain communication interface or protocol of the system under test. Since these adapters were written in C like the clients, they could implement non-trivial algorithms as desired. Each workload definition referenced the necessary adapters so that they were distributed to the clients by the master before starting the measurement.

While the resulting architecture was thus comparatively simple, one part of the application was given special care: The measurement strategies as the core of the measurement process were designed as state machines. Note that the state machines for load definition described above are of descriptive character and subject to be created and changed by the every-day users of SyLaGen, while the state machines for strategies we are now talking about are intended to be models for the implementation and thus created and edited by the developers. However, no modeling tools were used to derive the implementation from models systematically – at the time of the initial development, the related technologies were far less advanced than today. However, it was not desirable to implement the measurement process as large parts of sequential program code and loose all semantic information by this means. For this reason, a design pattern for state machines was employed that stored the basic information about states and transitions inside the program code. It followed some simple rules:

- A class `Priostate` was instantiated to represent single states. Each state instance was given a name to make logging messages comprehensible later on.

```
Priostate verify = new Priostate("Verify");

// ...

Transition foundMarkGoToVerify = new Transition("FoundMarkGoToVerify", verify) {
  boolean checkCondition() {
    return theState.loadTooHighOrNoMoreMeasurementsPossible();
  }

  void doAction() {
    phase="Verify";
    theState.saveResults();
    theState.decreaseToFirstUnexplored();
    theState.doMeasure();
  }
};
```

Listing 1: The informal program code pattern used in the first version of SyLaGen with a state and a transition.

- An interface with `get` methods provided access to the business logic of SyLaGen and allowed to extract variables related to the measurement, for example the number of restarts after errors.
- A class `Transition` was instantiated for each transition and connected to state instances. Each instance implemented two methods: `checkCondition` evaluated expressions related to method calls on the variables interface; `doAction` executed business logic by accessing other parts of the program code.

An exemplary part of the program code pattern can be seen in listing 1. In the upper part, a state instance is created. In the lower part, a transition instance is created. In order to supply it with guard and action program code, it is created as an anonymous class implementing both required methods.

The state machine was first populated by creating instances for states and transitions. Afterwards an execution method was called that repeated the following steps beginning with the start state: (1) Invoking and evaluating all `checkCondition` methods of the transitions emanating from the current state. (2) Selecting a transition whose guard returned `true` and invoking its `doAction` method. (3) Setting the target state of the chosen transition as the new current state.

By this means the state machine functionality was made explicitly and comprehensible for developers in the program code. However, no connection to any formal model was maintained, neither implicitly nor explicitly.

### 3. Review and Extensions

After the initial development, SyLaGen had been in use for several years both for industrial purposes and in academic courses. Besides minor maintenance changes, there was no critical review of the source code and no plans for further major development steps. However, even maintenance of the clients became difficult since the C-related tool chain was outdated and no experienced C developers were available to create a new one. To cope with this problem, the client was re-implemented in Java in 2006. At this time, the authors discovered that documentation for the initial development was partly incomplete or out-of-sync with the actual program code, which can be considered a quite common problem for long running projects. In addition,

the look and feel of the master's user interface had also become outdated and old fashioned, as well as the style of communication and data exchange between master and clients became laborious and frail in comparison to more up-to-date technologies. These were major issues since industry projects were still accomplished, although the project partners did not participate in the development, but only ordered measurements without knowing the underlying system.

Thus it was decided to start a complete re-engineering project for SyLaGen in 2007. In contrast to the initial development, the decision for re-engineering SyLaGen was a pure academic project without industrial partnership. Moreover, it did not happen inside a limited time frame or with dedicated resources. One the one hand, this allowed for more freedom and experiments during development, but on the other hand the available resources where generally low and could neither be increased by referring to upcoming project deadlines nor by additional funding. As a consequence, the first step towards a new version of SyLaGen was a student project that was concerned with design recovery and code review for the existing version of SyLaGen. This critical review resulted in two interesting facts: First, measurement strategies were implemented as explicit state machines as described in the previous section, but they were not as well decoupled from algorithmic details as it was required for the intended flexibility. Second, the behaviour of several other parts of the system could also easily be described as state machines, suggesting at least a far more modular architecture than the one that was implemented.

### *3.1. New Requirements and Re-engineering Goals*

Not only the code review and design recovery activities but also experiences from years of using SyLaGen set up the goals for a second version of SyLaGen: First, the architecture of the master should be modularized, implementing aspects like measurement control, master-client-communication and user interface in independent software components. Particular focus was put on decoupling the state machine descriptions for measurement strategies from execution details realized by the measurement control component. This requirement was not only driven by the academic goal of a fully modularized architecture, but also by the need for implementing new measurement strategies. A more modular architecture was also required in order to add sophisticated reporting capabilities which were not included in the initial version. Second, the old user interface and communication protocols should be replaced by new versions. The new user interface should be based on the Rich Client Platform of the Eclipse development environment [13] since it provides many features for building editors and is at the same time integrated in program code development tools, which is of interest for adapters. The communication protocol should make use of XML to wrap complex data structures, so that it could be handled with standard protocols instead of proprietary protocols and still stay flexible for extensions in the future. Replacing the old proprietary protocols in turn includes rewriting the communication interface of the clients as well. Third, the new capabilities gained from using Eclipse RCP should be used for a new input editor for formulating workloads in a more convenient format. However, this last point was more a nice side effect than a crucial argument in favour of re-development.

### *3.2. Implementation of the New Version*

The considerations for a redesign lead to a module structure that is shown in figure 2: The overall measurement is controlled by a user interface module and an attached configuration store. The main task of the user interface is the creation of the workloads for the performance tests, which are validated by a related module. When the measurement is started, the workload information is passed to the measurement module. It uses the client module which is connected to

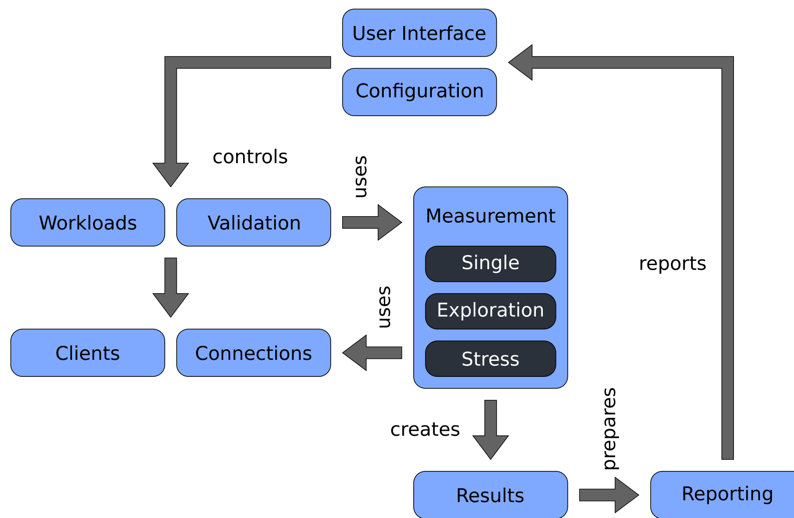



Figure 2: The module structure in SyLaGen with the basic kinds of relations between modules during the measurement process. The load generation strategies in the "measurement" module are embedded in and tightly connected to the different modules.

actual clients by means of a connection module that servers as an abstraction layer over different possible communication protocols. In the measurement module, the different load generation strategies are realized as plug-ins. They share the business logic provided by SyLaGen, but differ with respect to the number of measurements and the number of clients to be used. When a measurement is finished, the data collected by the clients is submitted to the result module. It is passed over to a reporting module that can employ different reporting technologies, e.g. writing the raw numbers to a file or alternatively passing them to a sophisticated reporting system. The prepared results are then provided to the user by the user interface again.

In order to facilitate a clean and formal approach to definition and implementation of the measurement strategies, a systematic re-engineering was applied to the existing source code. This re-engineering was supported by the fact that the state machines contained information about states, transitions, guards, and variables. The result of this was a set of state machines models defined in the timed automata model checker UPPAAL [8], which was chosen for its good visualization, simulation, and verification capabilities.

The other parts of the application were not developed based on models. The reason for this is the high number and tight connection of dependencies regarding platforms and frameworks:

- The user interface module completely depends on the structures determined by the Eclipse RCP platform. The resulting user interface can be seen in figure 3.
- The workload module depends on the XML schema of the desired workload format and frameworks to map them to in-memory object structures.
- The validation module depends on frameworks that can inspect compiled Java libraries in order to validate adapters.

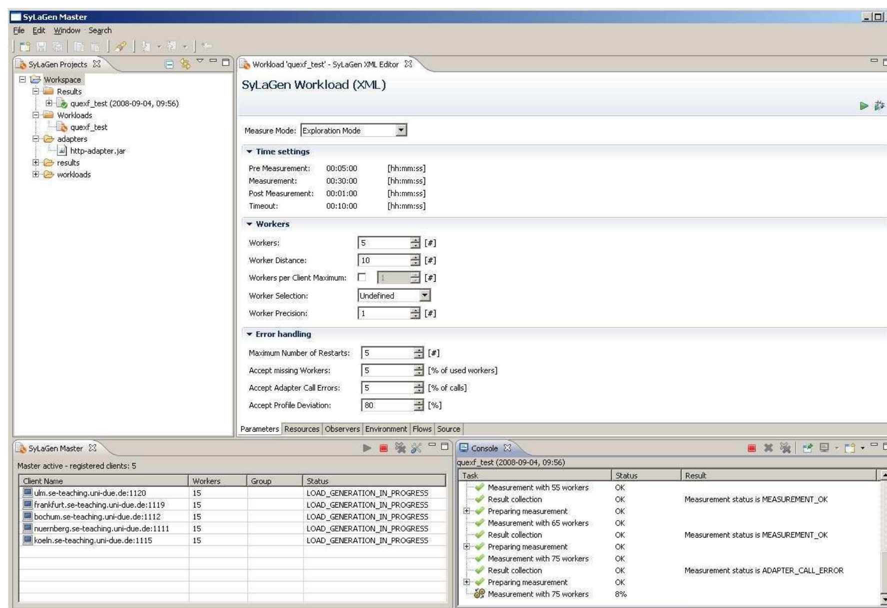



Figure 3: The user interface of the second version of SyLaGen based on Eclipse Rich Client Platform. We can see the project explorer on the left hand, a workload being edited in the middle, and the status display of clients and a running measurement at the bottom.

- The connection module currently uses a lean socket-based protocol to allow for connection to clients running on different platforms. For this purpose, all data is adjusted to text and number formats defined for this protocol.
- The measurement module provides calculations regarding the data used during measurement, for example mean throughput and request times.
- The result module embraces statistical calculations to merge results collected from single clients into a consistent overall result.
- The reporting module interacts with existing libraries for creating simple files, spreadsheet files, or input for reporting systems.

While for some of these tasks certainly model-driven approaches [4] can be applied, the number and variety of requirements precludes a consistent model-driven approach. A derivation of the implementation from models would either have been limited to few parts of the application or would have introduced a notable effort for customization of existing model-driven development tools. A generation of program code for the strategies would have been possible, but was not desirable since it would have been an isolated solution [7, 15].

However, an ad-hoc implementation was also not feasible. Since SyLaGen is an academic tool used in different projects and research contexts, requirements change often and quickly. Thus it was soon clear that it was not desirable to manually derive the implementation from these models again, since this would lead to the facts that the models (1) would just be a documentation and (2) would soon be inconsistent if no notable constant effort would be put into synchronizing

models and implementation. It was thus clear that possible approaches currently available did not fulfill the objectives to develop the software in a consistent manner while at the same time automating the synchronization with the models.

Following this conclusion and reflecting on the pattern-based solution from the initial version, we developed a new approach for implementing functionality based on models that can be integrated with arbitrary program code, as we will explain in more detail in section 4. At this time of the development it turned out as very beneficial to run the re-design project without industry partners and tight time frames, because this situation allowed us to delay active development of SyLaGen for experiments on the approach. As a result of these experiments, the second version of SyLaGen is as modularized as desired, uses all frameworks and platforms as necessary, and contains measurement strategies that are based on and formally connected to formal models.

## 4. Formalized Design Patterns

The need for a model-based development technology that integrates in a heterogeneous application like SyLaGen lead to the development of alternatives. It is likely that our research in this area would have been of different character if it was not motivated by the need to development SyLaGen with its different – and partly contradicting – requirements. Moreover, we are convinced that we would have needed more time to achieve practical results without this use-case-driven way of research.

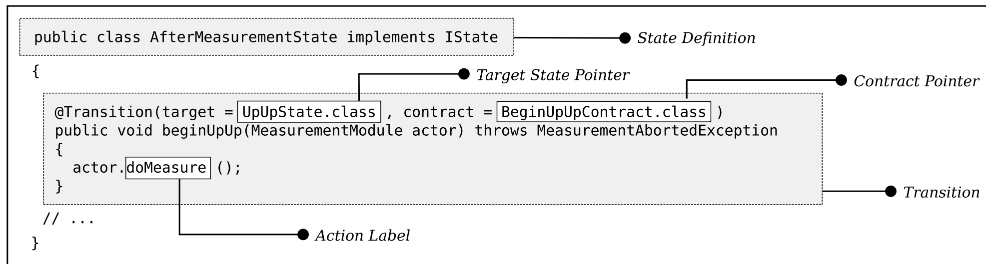
### *4.1. Approach*

The informal design pattern used in the first version of SyLaGen as shown and explained in section 2.2 was considered to be optimally integrated in the overall application. However, it lacked the possibility for a synchronization with formal models. The main reason for this is that it would not be possible to interpret the program code with respect to abstract models in an automated way: The fact that instances are created for single model elements makes it necessary to interpret possibly arbitrary algorithms to determine the model specification from object instantiations.

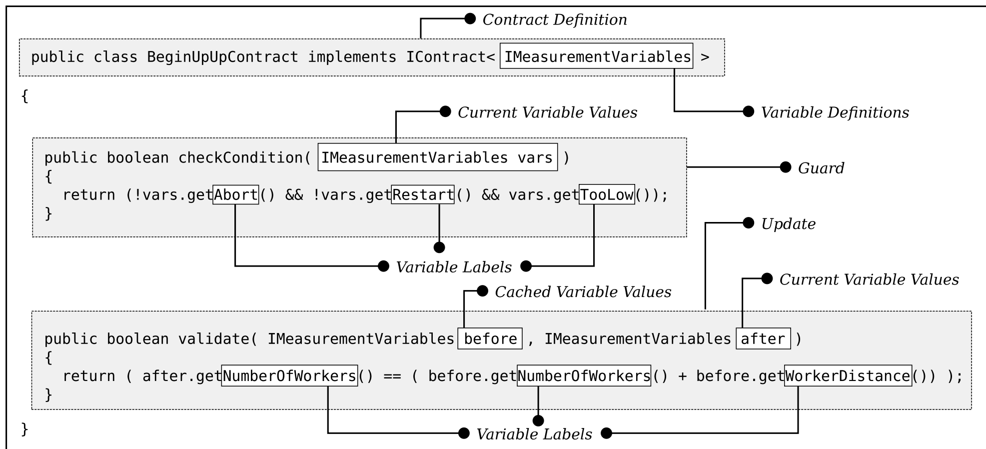
We therefore decided to modify the existing approach mainly with respect to the storage of information. For this purpose we use *attribute-enabled programming* [9] that uses meta data facilities introduced in programming languages like Java [12] to store meta data inside the program code and interpret the static structures by this means. This results in program code patterns that are formalized so that the static structures can represent the abstract syntax of models. Thus transformations can extract complete models from the code accordingly. The approach has so far been published as *embedded models* [2].

In SyLaGen, an embedded model for state machines based on UPPAAL has been created. It stores the complete state machine syntax in static elements of the program code and re-uses for this purposes some principles of the informal design pattern of the the first version, but also adds some additions:

- An interface provides `get` methods that abstract from the overall business logic of SyLaGen and provide a limited set of well-defined variables to be used in the state machine.
- Another interface (called *actor*) provides entry points to the business logic that can be called from inside the state machine. The names of the methods are by this means the action labels for the state machine.




**State and Transition Definition in Source Code**




**Contract Definition in Source Code**

Figure 4: A state definition with an outgoing transition and its contract. The first method of the contract checks a pre-condition with the current variable values, while the second method checks a post-condition by comparing the current values to previous values.

- States are represented as class definitions. The class name is interpreted as the name of the related state.
- Transitions are represented as methods inside state classes. They contain a set of method calls to the actor interface, so that each transition has a set of associated action labels.
- Meta data at transition methods refers to the target state class definition and to methods containing expressions for guards and updates. These expressions evaluate to boolean values and access the variables in the related interface.

An example of the formal program code pattern can be seen in figure 4. It shows a part of the SyLaGen state `AfterMeasurementState` in the “exploration” strategy that decides how to proceed after a single measurement was performed. As we can see, the single elements of the program code can be interpreted unambiguously with respect to the state machine model. Since the classes are not instantiated to be equipped with semantics, the program code can be statically analysed at development time and run time. Some more details have already been described in our previous publications [2, 1].

The execution of this pattern follows a similar principle as in the first version, but also with some important changes. These changes are based on the fact that our design pattern introduces a new abstraction layer and thus an additional layer of indirection and modularity. Consequently, execution is not based on object instances, but on the static structures instead, which are accessed by means of structural reflection [6]. Thus the state machine is not populated by creating instances of state and transitions inside the application source code itself, but by reading the program code by an execution framework in the measurement module. The execution framework for the embedded model interprets the program code as follows beginning with the start state's class definition and performing the following steps: (1) Instantiating the class, invoking and evaluating all guard methods of the transition methods in the current state. (2) Selecting a transition whose guard returned `true` and invoking its transition method. Besides the class instantiation at beginning of step 1, these steps are similar to what the old version did. (3) Invoking the update expression to determine if the business logic has performed changes that are compatible with the state machine specification. This step of permanent self-verification could now be introduced since the necessary information is now available from the embedded formal model. (4) Setting the target state of the chosen transition as the new current state until a final state is reached. This step again is similar to the last step in the old version. However, it has to be noted that all steps that are similar to the old version are now performed by the execution framework and not by the source code defining the state machine. Thus the goal of decoupling strategy definition and strategy execution can be considered fulfilled in the new version.

### *4.2. Usage for Development*

This complete information about the model allows not only to specify the state machine semantics more precisely than before and execute them in an automated way. Moreover, we can extract the model from within the program code and view or manipulate it in appropriate modeling tools, in this case UPPAAL as shown in figure 5 [10]. Since the program code pattern is a valid notation for the complete model specifications, we can also transform a changed model back to program code and thus apply changes.

At run time it is possible to monitor the execution of the program code pattern with respect to the state machines since the static structures of the program code are interpreted by the execution framework. A tool exists that shows the related model elements and their behavior, including a visual view of the state machine of the measurement strategy currently in use. By this means it is possible to track errors in the context of the abstract specifications without relying on meta data or tracing information, but only on the embedded model that is contained in the program code without any additional effort.

Thus the approach fulfills the needs of the academic tool development in this case:

- The number and variety of frameworks and platforms in use precludes solutions that require a complete formal definition at higher levels of abstraction or determine the software architecture by themselves. Instead, program code based on models is required to seamlessly integrate in the existing structures. The formalized design patterns allow for this since they share the notation of the program code that is also used by the frameworks and platforms.

- The high frequency of change requests makes a separate maintenance and synchronization of program code and complex and time-consuming. The formalized program code patterns works with different abstraction levels, but uses only one notation to store them, so that changes always affect all abstraction levels.

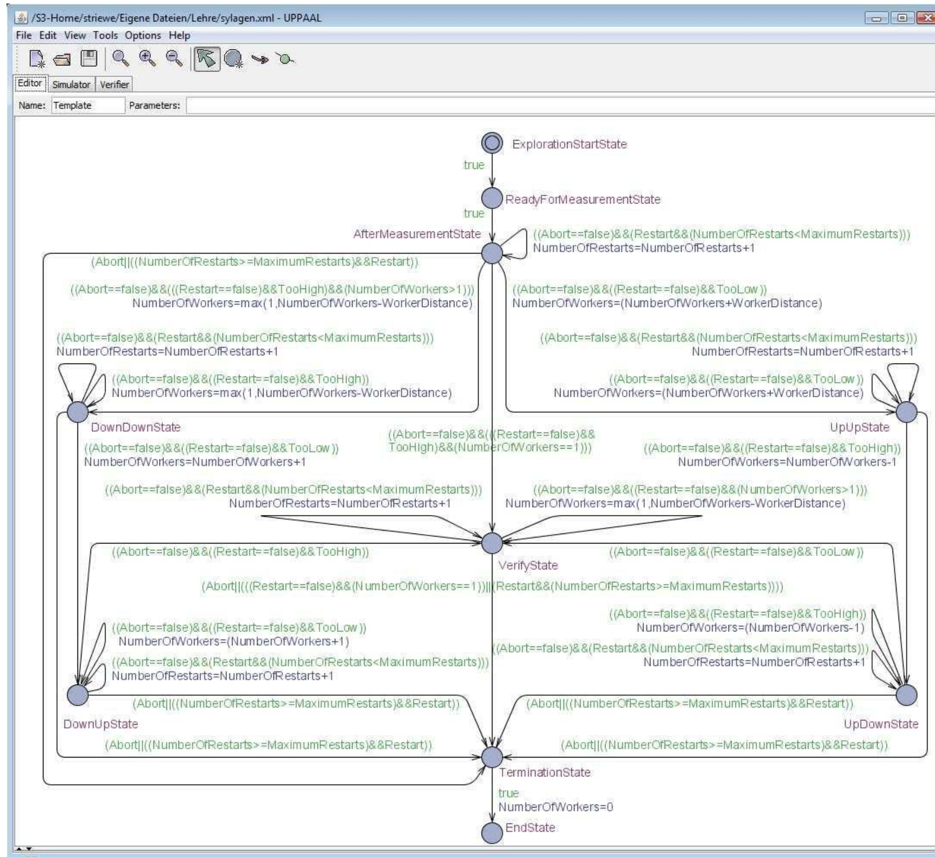



Figure 5: The UPPAAL model of the state machine controlling the "exploration" load generation strategy.

- Resources for industry-quality documentation are often not available in an academic context, so that there is a constant risk that the model documentation may not fully represent the reality. In the shared notation of the program code, documentation is always up-to-date since the model specifications can be extracted as long as the code complies to the formal specifications for the program code pattern. This can easily be validated or enforced with static code analysis.

In summary, the development and maintenance of SyLaGen was very interesting from our point of view: The reflection on the technologies in use and the requirements that occur for academic tools lead to a different model-based development approach, which has proved to work in this context since 2008.

## 5. Conclusion and Future Work

In this contribution we presented the development of the load generator platform SyLaGen. Considering the development of academic tools, it is from our point of view interesting with

respect to two aspects. First, we considered special (technical as well as organizational) requirements that lead to the need of non-standard solutions. In this case, certain quality criteria were desired, but not easy to meet in the dynamic context of academic projects. Second, the academic context lead to a solution that is from our point of view generally applicable to problems of the same class. This kind of research that happens beside actual projects, in similar contexts referred to as *serendipity* [14], is in our opinion a valuable contribution of the development of "real" applications in academia.

Future work regarding SyLaGen will include more reflections on model-based development. On the one hand, it is desirable to cover more parts of the application with formal models. This affects foremost the overall process of the measurement and also the modules and their interactions. Since it will not be feasible to generate the complete code, we want to create appropriate embedded models for these purposes. On the other hand, this leads to the questions how different domain-specific models can interoperate: By using the source code as the only notation, we will have a situation where different domain-specific models are tightly connected. A formal specification of their interoperability is desirable. SyLaGen as a tool is large enough to serve as a realistic and demanding case study for both kinds of research in model-based development.

## References


[1] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding State Machine Models in Object-Oriented Source Code. In *Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008*, pages 6–15, 2008.

[2] Moritz Balz, Michael Striewe, and Michael Goedicke. Continuous Maintenance of Multiple Abstraction Levels in Program Code. In *Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development - FTMDD 2010, Funchal, Portugal*, 2010.

[3] Reinhard Bordewisch, Bärbel Schwärmer, Michael Goedicke, and Peter Tröpfner. Lastsimulation für anwendungsumgebungen in vernetzten it-architekturen. *Mitteilungen der GI-Fachgruppe MMB*, (43), 2003.

[4] Alan W. Brown, Sridhar Iyengar, and Simon Johnston. A Rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.

[5] Tom Copeland. *Generating Parsers with JavaCC*. Centennial Books, 2nd edition, 2007.

[6] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[7] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[8] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[9] Don Schwarz. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. *ONJava.com*, June 2004. `http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html`.

[10] Michael Striewe, Moritz Balz, and Michael Goedicke. Enabling Graph Transformations on Program Code. In *Proceedings of the 4th International Workshop on Graph Based Tools, Enschede, The Netherlands, 2010*, 2010. accepted for publication.

[11] Michael Striewe, Moritz Balz, and Michael Goedicke. SyLaGen - An Extendable Tool Environment for Generating Load. In Bruno Müller-Clostermann, Klaus Echtle, and Erwin Rathgeb, editors, *Proceedings of "Measurement, Modelling and Evaluation of Computing Systems" and "Dependability and Fault Tolerance" 2010, March 15 - 17, Essen, Germany*, volume 5987 of *LNCS*, pages 307–310. Springer, 2010.

[12] Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™ Programming Language, 2004. `http://jcp.org/en/jsr/detail?id=175`.

[13] The Eclipse Foundation. Eclipse website. `http://www.eclipse.org/`.

[14] Pek van Andel. Serendipity: Expect also the Unexpected. *Creativity and Innovation Management*, 1(1):20–32, 1992.

[15] John Vlissides. Generation Gap. *C++ Report*, 8(10):12, 14–18, 1996.

# Industrialization of Research Tools: the ATL Case

Hugo Brunelière, Jordi Cabot, Frédéric Jouault, Massimo Tisi, Jean Bévizin

*AtlanMod, INRIA RBA Center & EMN, 4 rue Alfred Kastler, 44307 Nantes, France*
*{hugo.bruneliere,jordi.cabot,frederic.jouault,massimo.tisi,jean.bezivin}@inria.fr*

**Abstract**

Research groups develop plenty of tools aimed at solving real industrial problems. Unfortunately, most of these tools remain as simple proof-of-concept tools that companies consider too risky to use due to their lack of proper user interface, documentation, completeness, support, etc that companies expect from commercial-quality level tools.

Based on our tool development experience in the AtlanMod research team, specially regarding the evolution of our ATL model transformation tool, we argue in this paper that the best solution for research teams aiming to create high-quality and widely-used tools is to *industrialize* their research prototypes through a partnership with a technology provider.

*Keywords:* tool, ATL, prototype, industrialization, model-driven

## 1. Introduction

In software engineering, many research groups work on topics directly linked to (and sometimes also inspired by) real industrial problems. This research activity usually end ups implemented in a tool that serves as proof of concept for the developed techniques.

These tools are technically good and solve real problems but unfortunately most times they are largely ignored by software practitioners. Technical quality is just one of the many factors that companies analyze when deciding whether to take the risk of adopting a new tool. Examples of other factors that influence this decision are: user support, good documentation or usability aspects. The problem is that resources of research groups are very limited and cannot be usually devoted to work on this non-core research activities (also because funding for tool development is not exactly a priority for funding agencies). Because of this, research groups risk missing the opportunity of having a large user base for their tools along with the benefits that this brings to the table (e.g. empirical validation of their research, feedback, visibility, collaboration opportunities and so on). This is specially true in emerging software engineering areas with growing industrial interest but without a dominant tool/s monopolizing the market and where some of our research tools could make a difference.

We believe that Model Driven Engineering (MDE) is one of these areas. MDE is a software engineering paradigm based on the use of models as primary artifacts of all software engineering activities. Many of these activities imply model manipulations, usually implemented as model transformations. Therefore, availability of tools for specifying and executing model transforma-

tions is crucial. Right now, the most popular[1] model transformation language and toolset is ATL (AtlanMod Transformation Language [1]). Regardless the technical quality of ATL (comparing technical aspects of ATL with respect to other alternative tools, as graph transformation tools [2] or QVT-based tools [3] is not the topic of this paper) we believe its non-technical features have been one of the key factors of its success.

In this sense, the goal of this paper is present our experience in the development of a long-term sustainable model for ATL and the choices (both at the technical and business model level) we have made during this process. In particular, we will emphasize our industrialization approach for ATL (through a partnership with the technology provider Obeo [4]) that has permitted us to meet the needs of our large user base while at the same time committing our resources only to the research aspects of the language. This *business model* has proven to be feasible and beneficial both for the company and for ourselves as a research group.

The rest of the paper is structured as follows. The next section introduces the ATL tooling. Section 3 comments on the challenges of developing a commercial-quality level tool while Section 4 and 5 explain our best practices to overcome these challenges at the technical and business model level, respectively. Section 6 generalizes these results to other tools and, finally, Section 7 presents some conclusions and further work.

## 2. Overview of ATL

This section provides some background on the history (Section 2.1) and characteristics (Section 2.2) of the ATL model-to-model transformation language [1] and highlights the complexity of the tool as it stands today 2.3.

### *2.1. History of ATL*

ATL is a model-to-model transformation language created by AtlanMod [5], a joint research team between INRIA and Ecole des Mines de Nantes. ATL was first developed as part of the Phd Thesis of Frédéric Jouault [6] started in 2003.

Simultaneously to the definition of the language, a toolset with the same name to specify and execute ATL transformations was also implemented. This allowed the first concrete use of ATL on industrial scenarios that took place in 2004 in the context of the French project CARROLL/MOTOR, in collaboration with CEA and Thales. Then, the development of the language and tooling has continued, mainly as part of two consecutive European integrated projects called MODELWARE (October 2004 - September 2006) and MODELPLEX (September 2006 - February 2010) which provided many concrete industrial use cases of ATL in various and varied contexts. These use cases provided new research challenges that helped us to improve the language.

ATL is integrated in the Eclipse open source community. It joined the Eclipse-Modeling GMT (Generative Modeling Technologies) project in 2004 and then moved to Eclipse-Modeling M2M (Model-to-Model Transformation) project [7] in 2006. This was the effective recognition of ATL as one of the de-facto modeling standards in Eclipse. We believe this integration in the Eclipse foundation has contributed a lot to the development of the community and of the tool in general.

[1] At least among researchers, based on the number of works referencing and/or using ATL; there are plenty of industrial users as well but this is more difficult to quantify and compare against other tools

Finally, as we will explain in the next sections, the ATL ecosystem and the user community became too large to manage only with the resources available in the AtlanMod team. Therefore, in 2007, it was decided to start a collaboration with the Obeo company with the purpose of industrializing ATL. This situation is still going on today and has proved to be very beneficial for both AtlanMod and Obeo.

*2.2. Transforming models with ATL*

ATL is a model-to-model (M2M) transformation language. M2M transformations specify the production of target models from a number of source models. ATL enables developers to define how source model elements must be matched and navigated in order to create and initialize the target model elements. Roughly speaking, an ATL transformation is defined as a set of declarative rules. Each rule matches a subset of the source model and creates an excerpt of the target model. Pattern matching conditions are expressed in the OCL language [8].

Source and target models can be instance of different metamodels. A simple ATL transformation *Mt* (see Fig. 1 transforms a model *Ma*, which conforms to a metamodel *MMa*, into a model *Mb*, conforming to a metamodel *MMb*. Since in MDE everything should be considered, as far as possible, as a model, *Mt* itself is defined as a model conforming to the ATL model transformation metamodel *MMt*. All three metamodels conform to a metametamodel *MMM* such as MOF or Ecore.

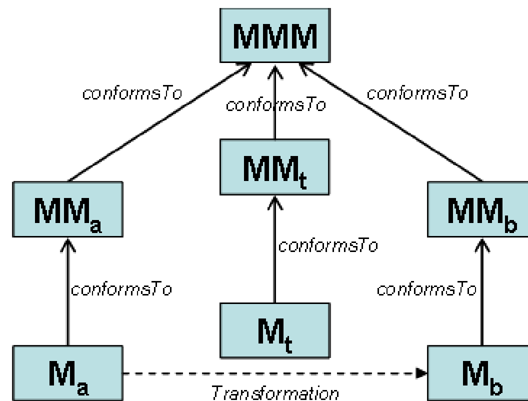



Figure 1: Model-to-Model transformation principle

As an example, Figure 2 shows an excerpt of the classic *ClassToRelational* model-to-model transformation example. The objective of this transformation is to transform a class diagram into a relational database schema, with the corresponding tables, columns, etc. In this example, the source metamodel is *Class* and the target metamodel is *Relational*. The actual transformation defines several rules (as *Class2Table* and *DataType2Type*) defining the mappings between the concepts from the two metamodels, and one helper method (helpers factorize ATL code that can be called from different points of an ATL transformation) navigating the input model to retrieve some data needed during the mappings.

Fig. 2 shows part of the *ATL Perspective*, with the *Project Explorer* presenting the different resources involved in the transformation, an excerpt of the ATL transformation displayed using the *ATL Editor* and the *Outline* of the transformation displaying all its important information.

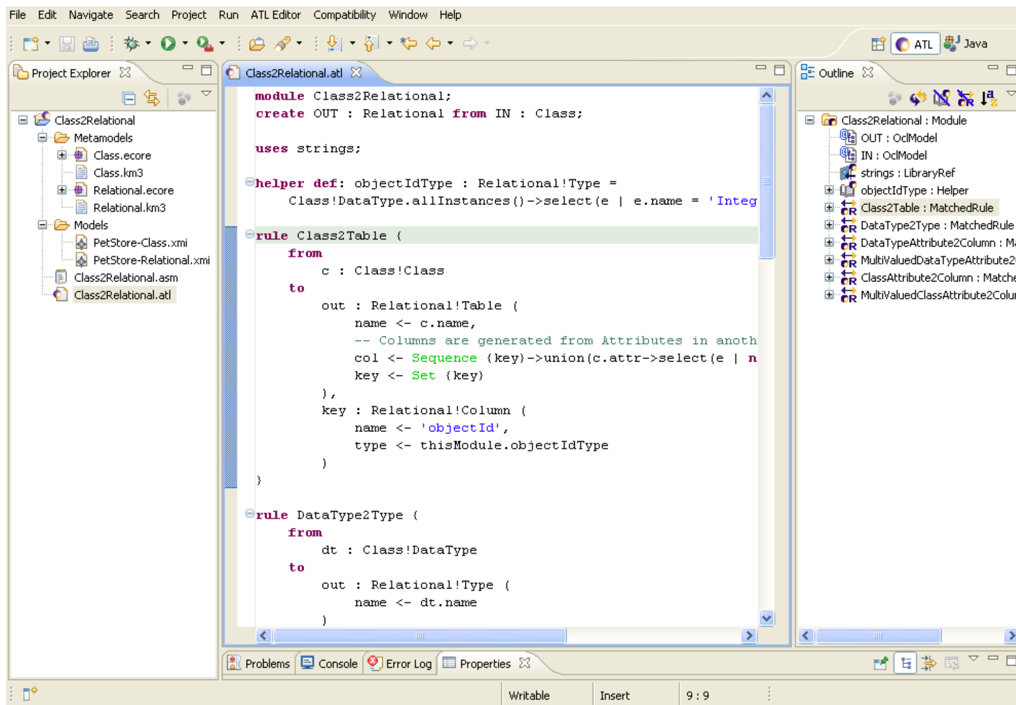



Figure 2: The ATL perspective with a simple transformation example

### 2.3. *Current status: the ATL project*

More than a simple model transformation language, ATL has evolved into is a full Eclipes project [9] directly under the umbrella of the Eclipse *Modeling* top-level project

As an Eclipse project, ATL proposes to the community a complete Eclipse IDE (Integrated Development Environment) coming along with the ATL language, core components and many different kinds of resources (see Figure 3). The ATL project is implemented as a set of specific Eclipse plug-ins and features. ATL has a dedicated Eclipse perspective, its own project nature and builders (and corresponding creation wizards) for ATL projects and files. An editor with extended capabilities (syntax highlighting, runtime parsing, compilation and error detection, code completion, etc) and corresponding outline, a debugger, profiler, launcher and ATL Ant tasks have also been implemented.

Apart from the regular plug-in downloads that can be get from the various update sites the project offers a large range of online information: conceptual overview, user guide, developer guide, technical specifications, etc. Many training resources like simple examples, tutorials or complete use cases are also available. Moreover, the project also provides a library of more than one hundred open source model-to-model transformations, all written in ATL and developed by various contributors, concerning many different metamodels and domains. These transformations can of course be taken as examples, but also reused as is or refined/extended for specific transformation scenarios.

Besides this, the ATL community is continuously expanding. There are already several external tools using ATL as their standard model-to-model transformation technology and, in addition

Figure 3: The ATL project on Eclipse.org

to the various publications and presentations related to ATL in major events, project committers and contributors regularly give support to ATL users via the dedicated Eclipse forum, Eclipse Wiki or Eclipse Bugzilla. For instance, around 40 different topics (i.e. discussion threads) directly concerning ATL have been active in the Eclipse-M2M forum just in the two first weeks of July 2010.

## 3. Challenges in the Development of a Commercial-Quality ATL

As ATL was growing into the nowadays complex ecosystem (see previous section) it was clear that ATL had the chance to become the reference model-transformation solution for companies starting to adopt MDE as their development paradigm. As a research group, this was very appealing since it would allow us, among many other benefits, to: 1 - Empirically validate our transformation language and, in general, our research ideas; 2 - to provide good on-the-field experimental data; and 3 - to increase the AtlanMod team visibility, increasing the number of companies/groups that could discover (and become users of) our other works in the MDE field and opening opportunities for future collaborations (e.g. technology transfer contracts or joint participation in international projects).

However, we quickly realize that it would be very difficult to maintain all the components of the ATL project and give proper support to users only with the resources available in the team.

Furthermore, focusing on providing a good technical solution would not be enough to keep and expand our user base, for potential users (specially companies) technical quality is only one of the many aspects they take into account when selecting a tool. ATL users started to ask for:

- better user support (e.g. responsiveness to questions, bugs and feature requests)
- up-to-date and complete tool documentation
- good user experience (ergonomy)
- long-term perspectives for the tool
- frequent upgrades and interoperability with the latest versions of other common used tools
- extensibility, adaptability, and all the other typical -ities
- backward compatibility (which many times is difficult to combine with innovative research)

In fact, we agree that all these aspects are important (as first users of the tool we would also benefit from them, e.g. when hiring a new team member a proper documentation set would ease his/her integration in the team activities) and they have been always in the wish-list of ATL though hardly ever made it to the actual to-do list. The main reason is that the low availability of funding for pure tool development forces us (and, in general, all research groups) to focus our resources on the most innovative and research aspects of ATL neglecting all these other issues. This is a direct consequence of the fact that most (all?) government agencies evaluate research groups (and their members individually) exclusively based on the number of papers published by the group, ignoring the impact in the community of the tools developed by them. This makes extremely difficult to convince young researchers pursuing a permanent position to dedicate time to any task that does not immediately and directly lead to new research results.

To overcome this situation, we have put in practice two kinds of strategies during the development of ATL. First, the design of the tool itself simplifies some of these problematic aspects and, more importantly, tries to facilitate the collaboration and technological transfer among all external actors contributing to the ATL development. The second strategy deals with the problem of convincing external people to contribute to ATL. In fact, given the open source license[2] of ATL, one could have the (naive) initial assumption that it would be easy to find external contributors ready to jump in and start working on all aspects a research group cannot cover. However, this assumption fails because few projects are succesful involving external contributors [10] and most times, these external users also prefer to work on the most challenging tool features and not, for instance, write documentation or improve the graphical user interface. For instance, many important ATL contributors are members of other research groups that extend or improve ATL as part of their research interests.[3] Therefore, we realized that, apart from having a good tool design, we needed to define a *business model* for ATL to guarantee a continuous influx of resources to satisfy the requests of our users while allowing us to keep focusing on the core research aspects of the tool. Section 4 and Section 5 elaborate on these two kind of strategies.

[2] More specifically, ATL follows the Eclipse Public License

[3] See, for instance, the program of the two ATL workshops organized so far: http://www.emn.fr/z-info/atlanmod/index.php/MtATL2009:Program and http://www.emn.fr/z-info/atlanmod/index.php/MtATL2010:Program

## 4. The ATL Solution: Technical Aspects

ATL tool design has been influenced by the principles of modularity, interoperability and use of standard (de facto) technologies as a way of facilitating contributions from independent actors taking part in the ATL development. Apart from this, becoming intensive users of your own tool is also a *best practice* to test if the previous design priciples have been properly followed.

### *4.1. Modularity*

The ATL framework has a modular structure based on the extension mechanism of the Eclipse platform [11], that in turn is built upon the OSGi open modular system. The *core modules* of ATL are shown in Figure 4. These modules are in charge of the operations that lead to the transformation execution: code parsing and static checking, compilation and bytecode interpretation.

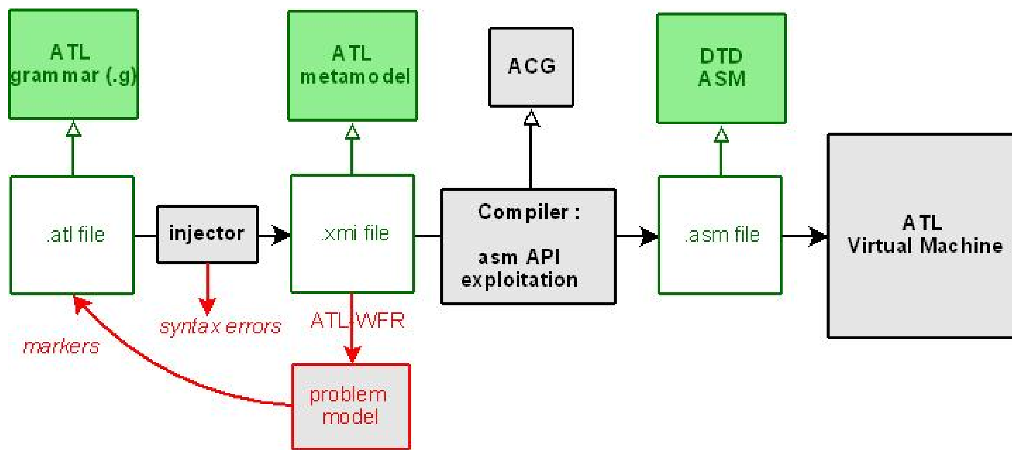



Figure 4: Components involved in the execution of an ATL transformation

The first component is the ATL parser (i.e. the *injector* in the figure) which takes an ATL source file as input and produces a model that conforms to the ATL metamodel as output plus a problem model storing all errors detected in the code. This error information is then used by the ATL editor to display the corresponding error markers in the ATL source file. The second component is the ATL compiler in charge of converting the ATL model into bytecode (ASM format) interpretable by the ATL Virtual Machine (VM). It has been implemented using ACG, a compiler-oriented DSL [12]. The last core component, the ATL VM, takes the compiled transformation and actually executes it on the specified input model(s) to generate as a result the output one(s). Additional ATL modules are implemented on top of these core ones to provide the main functionalities of the ATL development environment (e.g., editor, debugger, profiler, launcher). Communicaton between the modules is done thanks to the standard Eclipse extension interfaces. These same interfaces can be used by external developers wanting to make use of ATL in their own Eclipse plug-ins (e.g., AMW [13]).

This modularization of the ATL core allows for a clear separation of concerns among the basic components in charge of the transformation processing. This decoupling has proven to be a valuable asset for collaborative work over the ATL core. For instance, it is common for independent research groups to locate a research problem in only one of these components. Thanks

to the ATL modularization the group only needs to get familiar with a limited set of technologies (e.g. ACG for the compiler) and can provide an updated interchangeable version of the module addressing that specific research issue. Examples of this are the optimization of the collection handling approach of ATL by updating only the virtual machine [14] or the development of an incremental execution mode for ATL by changing only the ATL compiler [15].

The modularization of ATL has shown its importance in several projects during the last years. As a further step in this direction, we plan to introduce a new fine-grained extension system for the ATL core components by defining a new set of extension-points in the core, i.e. of functionalities that can be delegated to externally provided plug-ins.

*4.2. Standard technologies*

An important driver in the success of ATL has been the selection of the right core technologies for grounding the tool. The search for an open framework, supported by a solid open-source community, and that could become the de-facto standard platform for open software engineering tools, lead us to choose the Eclipse platform. The growing popularity of Eclipse and the increasing number of tools migrating to this platform proves our decision right and increases the opportunities of collaborations between ATL and the other Eclipse projects.

A first benefit of integrating ATL in Eclipse is that Eclipse relieves the ATL project from implementing most of the components of the IDE. The ATL perspective reuses many of the standard IDE components provided by Eclipse and adapts them to the specific needs of ATL. Eclipse also provides an extensibility mechanism that facilitates the definition of ATL extensions. Eclipse plug-ins can define *extension-points*. An extension point indicates that the plug-in is able to delegate that specific functionality to a different plug-in. *Extension-points* can be answered by *extensions*, that implement an interface and descriptor matching the extension-point. Availability of a standard extension mechanism is a key factor in enabling the creation of re-usable components from a complex and heterogeneous community.

Finally, the success of a model transformation technology like ATL has necessarily to rely on the success of the underlying modeling framework it uses (i.e. if models to be used as input/output of ATL transformations had to be conformant to a modeling framework nobody uses then, obviously, few people would use ATL). While the modular architecture of ATL favors interoperability and adaptation to different frameworks (see next subsection), we decided to privilege[4] the Eclipse Modeling Framework (EMF) since it soon appeared that among several possible alternatives, EMF stood out as the de facto standard solution for model handling. The fact that today most model-based tools in Eclipse work with EMF facilitates the selection of ATL as a model transformation language for their model manipulation activities.

*4.3. Interoperability*

While the early choice of Eclipse and EMF has certainly influenced the success of ATL, the design of its architecture has followed the principle of maximum independence from specific technologies so that we can change these technological choices in the future. For instance, as a transformation language, ATL should be independent on the particular way a model is represented and handled in the modeling environment.

[4]We say that EMF is a privileged modeling framework in the ATL toolkit because there is an optimized version of the virtual machine available for EMF that guarantees the maximum efficiency when working with EMF models

On the one hand, this criterion guarantees that the longevity of ATL will not be influenced by a particular technology (e.g., EMF) going out of fashion. This is particularly important in an industrial context, where the preservation of investments is a central requirement. On the other hand this independence enables a straightforward path for exporting ATL to other modeling environments (e.g., Microsoft DSL Tools [16]), with the opportunity of remarkably expand the user base.

As technical solution for remaining independent from the modeling layer we introduced the notion of virtual machine to execute the ATL compiled bytecode. The ATL Virtual Machine is an abstract computing machine, associated with its own instruction set and specialized in model manipulation. Within the virtual machine, an intermediate component called Model Handler Abstraction Layer deals with the specifics of the modeling layer. This component translates the instructions of the VM for model manipulation into instructions for a specific model handler. By implementing an ad-hoc model handler it is possible to use the same VM on top of various model management systems (e.g., Eclipse EMF, Netbeans MDR, Microsoft DSL Tools) [17].

Moreover, porting the VM itself to other runtime environments (e.g. from Java to .NET) opens the door to immediately execute ATL transformations in them since any compiled ATL transformation can be indifferently executed on all implementations of the ATL VM that conform to the provided specification, independently of the underlying technology [17].

### *4.4. Eating your own dog food*

We believe it is important that research groups are the first users of the tool and that the tool is bootstrapped when possible. For instance, the ATL compiler towards ASM bytecode is written in the ACG DSL. The ACG compiler is bootstrapped, being itself written in ACG and being parsed, compiled towards ASM and executed using the same technology (the TCS parser, ACG compiler, ASM virtual machine) used to parse user defined ATL transformations. In the same way the execution of ATL transformations is partially performed using other ATL transformations (e.g. to generate the problem model).

Apart from being a non-trivial test for the tool, the main benefit of this is that any improvement on the tool components improve not only the user-defined transformations but also the ATL engine itself.

## 5. The ATL Solution: Business Model

The software engineering best-practices that have been followed since the first years enabled, in a first phase, the construction of an initial open-source community around the tool under the guidance of the research group AtlanMod. However, as argued in Section 3, once the user base started to grow and the ATL project became more complex this was not enough to guarantee a durable and expanding environment for ATL. A novel business model was required, to coordinate needs and efforts of the actors operating around ATL.

In 2007, the industrialization process for ATL started. AtlanMod partnered with Obeo [4] to ensure the existence of an open source but commercial-quality version of ATL. As part of this agreement, Obeo started committing resources on the non-core aspects of ATL (the amount of resources is more or less equivalent to the ones AtlanMod devotes to ATL research topics).

From the point of view of the research group, the main goal of the agreement is allowing AtlanMod to focus its resources on new research activities, and letting Obeo take over traditional software development and maintenance tasks, including performance and usability improvements, bug fixing and user support. Thanks to this agreement, ATL technology reaches an

industrial quality, becoming a very attractive tool for company users and a stable base for new research directions.

Obeo, on the other hand, gets to strenghten its presence and visibility within the Eclipse community and among industries using its open-source tecnologies. This strategic positioning is used to create a network of key partner companies that help to expand the Obeo market. As part of the agreement, Obeo also becomes the lead ATL training company (AtlanMod redirects to Obeo all ATL-specific course requests coming from companies) and can deliver adapted versions of ATL to specific companies with special needs. Moreover, ATL complements well the other MDE technologies in the the Obeo offering (that already include, for instance, the Acceleo tool for model-to-text transformations) and it is already been used as a key component in Obeo's new tool releases. This in-house development of ATL and the privileged relationship with the creators and lead research contributors of ATL (ourselves) give Obeo a competitive advantage over other companies that also propose products based on ATL.

More specifically, and according to the agreement with AtlanMod, Obeo takes responsibility over a precise set of activities around the ATL technology:

**Quality assurance.** The main responsibility of Obeo is in maintaining the ATL software at a quality level that is acceptable for wide industrial use. This task is twofold, since it includes a *reactive* part, mainly related to managing bug reporting and correction, and a *proactive* part, for the optimization of the ATL codebase, with a special attention to scalability.

**Interoperability.** The ATL tool is intended as a part of a complex modeling system, composed both by modules built within Obeo itself (e.g., Acceleo) and tools provided by third parties (e.g., modelers, modeling frameworks). Obeo takes care of the issues connected with the interoperability among ATL and the related modules, and implements new solutions to improve this interoperability.

**Continuity.** In an industrial context it is important to guarantee that the evolution of development tools will not unexpectedly break existing projects. The integration in ATL of new technical solutions, coming from Obeo or academia, has to be performed in an incremental way, carefully preserving backward compatibility with previous versions.

**User experience.** One of the priorities of the industrialization of ATL is the improvement of the general user experience in the development environment. Obeo makes use of ergonomy principles for the improvement of the IDE, and adds typical facilities such as wizards and hints. The company also takes care of the internationalization of the user interface.

**Release management.** Obeo takes charge of the management of ATL releases, by defining milestones, building, testing, packaging and distributing new versions.

**User support.** Finally Obeo offers support to the user base, in both the forms of free and subscription-based support. Casual users are assisted for free by managing the newsgroup and the mailing list on the Eclipse website. Obeo also commits in extending and updating the documentation of the ATL components. On the other hand Obeo has an immediate income from offering industrial users a personalized support and training programs.

At the same time Obeo took the lead of the official (stable) version of ATL, a new branch of ATL was formed, *ATL Research*, to host the experimental ATL versions, under the supervision of AtlanMod. Figure 5 illustrates the mechanism for technological exchange between the two

branches of ATL. The official versions of ATL, maintained by Obeo, are publicly distributed and constitute the baselines upon which research prototypes are built. These experimental *ATL Research* prototypes exhibit novel solutions for particular research projects. In the cases in which these solution are considered interesting for the general public, they are integrated in the subsequent official version of ATL. Depending on the stability of the prototype and on its compliance with a set of quality criteria (e.g., scalability, backward compatibility) the module in ATL Research can be directly imported in ATL, or, more commonly, its functionality is re-engineered by Obeo before being integrated in the stable branch. We believe this two-branch strategy is a good solution to avoid any restrictions on the research activities around ATL while not damaging the stability and backward compatibily properties desired by industrail users.

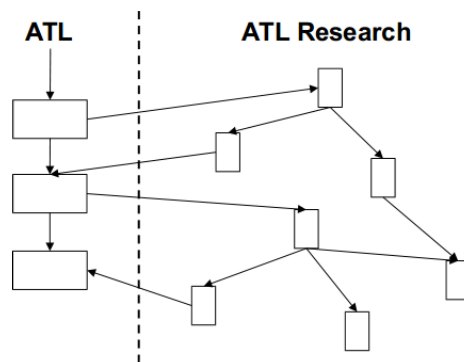



Figure 5: ATL and ATL Research

We remark that the initial choice of licencing every part of ATL under an open-source licence (this includes both the ATL research branch and the official branch maintained by Obeo) has remarkably simplified the creation of this new business model. The open source licence automatically guarantees the possibility to transfer every ATL update between the company and the research lab. Moreover this open source licence also assures to any third party, especially other research groups, that this code exchange will not benefit only AtlanMod, but the whole research community . This guarantee has the effect to stimulate the contribution of new code from ATL users and researchers, with a return gain for AtlanMod and Obeo themselves.

## 6. A Sustainable Open Source Business Model for Industrializing Research Tools

We are now replicating this industrialization process with two other MDE technologies from the group: the software modernization tool MoDisco [18] (together with Mia-Software [19]) and, more recently, our Megamodeling model management approach [20] (with ProDevelop [21]), with the same levels of success and satisfaction for the results of the collaboration achieved so far. Therefore we plan to generalize this strategy in most of the new projects of the team.

In this sense, this section sketches our *sustainable open source business model for industrializing research tools* triangle that summarizes the general industrialization schema we envision. There are three main actors in this triangle: the research lab, the technology provider and a big company/user community (see Figure 6). The triangle allows connecting research groups and industrial users and match their interests.

The triangle works as follows:

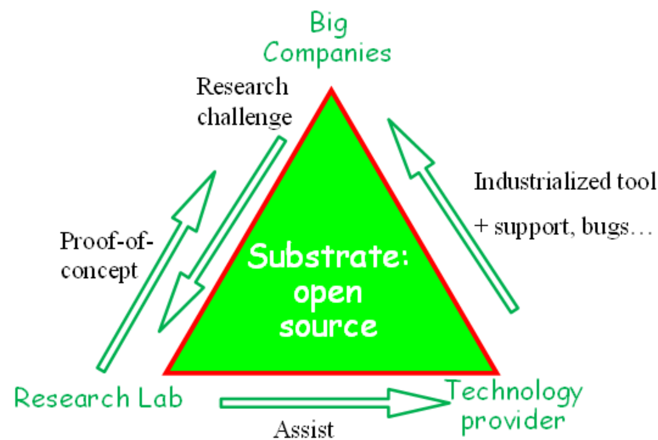



Figure 6: Open source triangle for the successful industrialization of research tools

1. Big companies (or big user communities) describe a challenging problem they face in their domain and that they would like to see solved (e.g. the industrial use cases that helped to drive the development of ATL in its early beginnings)
2. The lab evaluates the problem and decides whether it is a relevant research problem
3. If it is, the lab conducts the research (and publishes papers about it) and implements a proof-of-concept to present the results to the big company
4. The big company evaluates these results and decides whether to go forward and request an industrialized version of the tool to use in its day-to-day practice
5. In that case, the big company, with the help of the research lab, selects a technology provider.
6. The research lab assists the technology provider during the industrialization of the tool
7. The technology provider releases the tool

The fact that this is an application-driven research (i.e. the starting point is a real problem that a big company wants to be solved) ensures the return of investment for the technology provider. Adopting open source as the common denominator in all these activities is not absolutely mandatory but facilitates a lot the communication between the different actors and maximizes the benefits of the relationship (e.g. for the research group it is easier to publish papers about it and the technology provider could commercialize services and adaptations on top of the tool for other big companies sharing the same problem). This triangle can also be used to characterize many other interesting scenarios between the actors. For instance, when the research group is not interested in the problem expressed by the big company, the company can still search for a technology provider to solve that problem. Detailing all possible relationships cannot be done here due to space limitations.

This idea of intertwining research groups and industry partners has been already praised for its benefits [22]. Similar to the *industry-as-laboratory* approach presented there, we believe that continuous communication and technology transfer, between research lab and industry, is very benefitial for both partners. However, we believe that our triangular structure can help overcoming the limitations identified in [22] (such as the short-term/long-term and evolutionary/revolutionary different interests between the lab and the industry), since it gives to the re-

search lab the flexibility to combine the benefits of working with real end-users and technology providers at the same time in a coherent global schema.

We strongly believe that this schema can be successfully adopted by other research groups as a strategy to develop industrial quality level tools with all the benefits that this brings to the group.

**7. Conclusion**

We have presented our strategy for developing high-quality tools that can become widely used by the software engineering community: industrialization of the research prototypes thanks to the partnership with a technology provider. We believe this is the best solution to ensure that somebody external to the team (in our case, this technology provider) is in charge of all non-core research aspects of the tool development (documentation, usability, user support, etc). These aspects are key to convince external users (specially companies) to adopt our tools but cannot be taken in charge by the research teams themselves due to their limited resources. We have validated this strategy during the development of our ATL tool and we are now replicating it with other tools of the group.

We believe our experience can be useful to other research groups willing to go from proof-of-concept tools to real industrial-level tools, with all the benefits this may bring to the team (in terms of visibility, feedback, project opportunities and so on). We also hope our example helps to convince as well technology providers and development companies about the advantages of participating in such joint initiatives; there is a viable business model behind it, as companies like Obeo or MIA-Software have demonstrated.

As further work, we would like to formalize a protocol for the establishment and execution of this kind of partnerships so that roles, resources and responsibilities of each participant are clear from the beginning. This will facilitate explaining this open source industrialization concept and developing new partnerships in the future. This is being investigated as part of our contribution to the OPEES European project [23] aimed to ensure long-term availability of innovative engineering technologies in the domain of dependable / critical software-intensive embedded systems.

Additionally, we plan to create a web-based collaboration platform for ATL extensions (we tag this project *ATL labs* due to its similarity with existing initiatives as the recent EclipseLabs.org proposal) to facilitate the autonomous experimentation and contribution by other people beside our main partner. This project will provide a sandbox where researchers can experiment with ATL and extend it for specific purposes. Such specialized extensions cannot be done on the official ATL branch since there the contribution policy is quite rigid and they don't fit in the tree structure of the ATL Research branch. ATL Labs is meant to host a number of orthogonal ATL extensions for specific needs that potential users can select and compose to get a specific ATL "flavor" adapted to their needs.

Finally, we plan to start using social networking tools to build a community of ATL users that can easily share their experience, help each other and participate in discussions about the future of ATL; the limited participative mechanisms in Eclipse (mailing lists, bugzilla, forums, ...) are not enough for this.

*Acknowledgments.* The present work has been partially supported by the ITEA2 OPEES European project.

## References


[1] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Sci. Comput. Program. 72 (1-2) (2008) 31–39.

[2] J. de Lara, H. Vangheluwe, Defining visual notations and their manipulation through meta-modelling and graph transformation, J. Vis. Lang. Comput. 15 (3-4) (2004) 309–330.

[3] OMG, MOF 2.0 Query/View/Transformation specification (2007).

[4] The Obeo company, `http://www.obeo.fr`.

[5] The AtlanMod team (Ecole des Mines de Nantes & INRIA), `http://www.emn.fr/z-info/atlanmod`.

[6] F. Jouault, Contribution to the study of model transformation languages, Ph.D. thesis, University of Nantes (September 2006).

[7] The Eclipse Model-to-Model (M2M) project, `http://www.eclipse.org/m2m`.

[8] OMG, UML 2.0 Object Constraint Language specification (2006).

[9] The Eclipse ATL project, `http://www.eclipse.org/atl`.

[10] S. Krishnamurthy, Cave or community? an empirical examination of 100 mature open source projects, First Monday 7 (6).

[11] Eclipse Helios Simultaneous Release, `http://www.eclipse.org/helios`.

[12] ATL VM Code Generator (ACG), `http://wiki.eclipse.org/ACG`.

[13] M. Del Fabro, J. Bezivin, F. Jouault, E. Breton, G. Gueltas, AMW: a generic model weaver, in: Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles, 2005.
URL `http://idm.imag.fr/idm05/documents/11/P11.pdf`

[14] J. S. Cuadrado, A proposal to improve performance of ATL collections, in: MtATL2010, 2010.

[15] F. Jouault, M. Tisi, Towards Incremental Execution of ATL Transformations, in: International Conference on Model Transformation, ICMT2010, 2010.

[16] Microsoft Domain-Specific Language (DSL) Tools, `http://msdn.microsoft.com/fr-fr/library/bb126235.aspx`.

[17] H. Bruneliere, J. Cabot, C. Clasen, F. Jouault, J. Bézivin, Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools, in: T. Kühne, B. Selic, M.-P. Gervais, F. Terrier (Eds.), ECMFA, Vol. 6138 of Lecture Notes in Computer Science, Springer, 2010, pp. 32–47.

[18] MoDisco, `http://www.eclipse.org/MoDisco/`.

[19] Mia-Software, `http://www.mia-software.com/`.

[20] M. Fritzsche, H. Bruneliere, B. Vanhooff, Y. Berbers, F. Jouault, W. Gilani, Applying megamodelling to model driven performance engineering, in: ECBS, IEEE Computer Society, 2009, pp. 244–253.

[21] ProDevelop, `http://www.prodevelop.es/`.

[22] C. Potts, Software-engineering research revisited, IEEE software (1993) 19–28.
URL `http://www.computer.org/portal/web/csdl/doi/10.1109/52.232392`

[23] Open Platform for the Engineering of Embedded Systems, `http://opees.org/`.

# Disnix: A toolset for distributed deployment

Sander van der Burg, Eelco Dolstra

*Delft University of Technology, The Netherlands*

**Abstract**

The process of deploying a distributed system in a network of machines is often very complex, labourious and time consuming, while it is hard to guarantee that the system will work as expected and that certain non-functional requirements from the domain are supported. In this paper we describe the *Disnix* toolset, which enables automatic deployment of a distributed system in a network of machines from declarative specifications and offers properties such as complete dependencies, atomic upgrades and rollbacks to make this process efficient and reliable. Disnix has an extensible architecture, allowing the integration of custom modules to build a distributed deployment architecture that takes non-functional requirements of the domain into account. Disnix has been under development for almost two years and has been applied to several types of distributed systems, including an industrial case study.

*Keywords:* Software deployment, Distributed systems, Service oriented systems

## 1. Introduction

The software deployment process of a distributed system, which consists of steps such as building components from source code, transferring the components from producer side to consumer side and the activation of the system, is usually a very difficult, expensive and time-consuming process. In many cases, there is a lot of labour involved, it is hard to guarantee that a system will work as expected, the system may break completely and it is a process which cannot be performed atomically (i.e. during upgrading a user may observe that the system is changing). Furthermore, there are non-functional properties from the domain that must be supported, such as security, privacy and performance.

In order to reduce the complexities in the software deployment cycle, several solutions have been developed. Many of these tools are specifically designed for component technologies such as Enterprise Java Beans [1], or designed for environments such as Grid Computing [2]. Furthermore, some general approaches have been developed, such as [3].

While these tools are useful for its purpose, some distributed systems are not homogeneous (i.e. not implemented by a single component technology or for a single platform/architecture). For instance, many Java EE systems are composed of components implemented in the Java programming language, but also of non-Java components, such as databases, which cannot be completely and reliably deployed by existing deployment tools.

*Email addresses:* `s.vanderburg@tudelft.nl` (Sander van der Burg), `e.dolstra@tudelft.nl` (Eelco Dolstra)

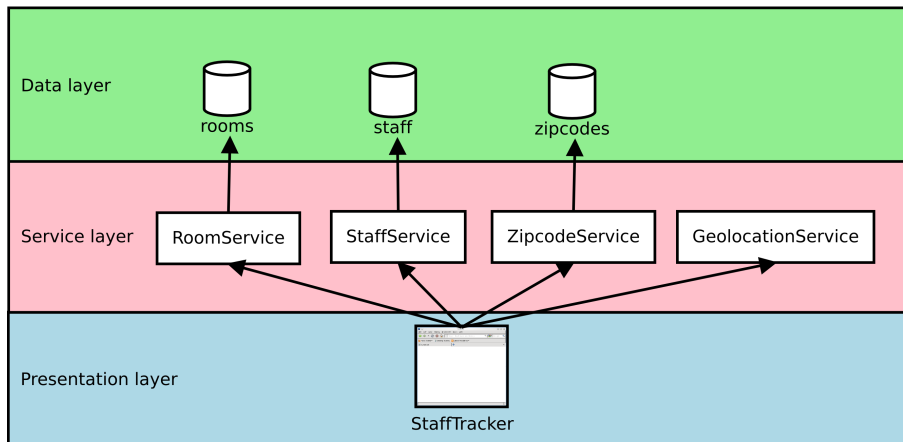



Figure 1: Architecture of the StaffTracker example system. Objects represent distributable components. Arrows represent dependency relationships between components.

Therefore we have developed Disnix, a toolset built on top of the Nix package manager [4, 5] used to automatically deploy a distributed system in a network of machines. In contrast to other tools for distributed deployment, Disnix is designed to support the deployment of complete *heterogeneous* systems, which are composed of components using various technologies on various platforms/architectures. Moreover, Disnix offers features to make the deployment process safe and reliable, such as *models* to describe services and machines in the network to *automatically* perform deployment steps, *complete* dependencies, *atomic* upgrades and rollbacks, *garbage collection* to safely remove obsolete components and a *modular* architecture to integrate with the domain in which the system is to be used. We have applied Disnix to several *use cases*, such as a service-oriented system designed for hospital environments [6].

In this paper we explain the concepts underlying Disnix, as well as its architecture, which consists of primitives to perform deployment steps in a generic manner and is *extensible* to make the deployment process suited to the target domain in which the system is to be used.

The paper is organised as follows. We first introduce a motivating example in Section 2. We then provide some background information in Section 3. Section 4 briefly discusses the purpose and concepts of the Disnix toolset. In Section 5, we explain the architecture of Disnix, such as its tools, libraries and design choices. Section 6 explains some extensions built on top of the Disnix toolset to make deployment more convenient for specific domains. Section 7 explains our experiences with the development and testing of the toolset, an open source community in which Disnix is being developed and some use cases. Finally, Section 8 concludes and outlines future work.

## 2. Motivating example

Figure 1 shows the architecture of the StaffTracker system, a simple distributed system developed as one of the example cases for Disnix that can be publically used. The StaffTracker is a system to manage staff of a university, and uses several web services to look up the location of a

staff member from an IP address, a staff member's zipcode from a room number, and an address from a zipcode.

The architecture of the StaffTracker consists of three layers. The *data layer* contains components that actually store data in MySQL databases. The *service layer* contains web services, providing an interface to the data stored in the databases (the GeolocationService uses GeoIP [7] to track the country of origin from an IP address). The *presentation layer* contains the StaffTracker web application front-end, which can be used by end users to manage staff of the university. This application invokes the web services in the middle layer to retrieve records or to modify the data stored in the databases.

All the components in Figure 1 are *distributable* components (which we call *services* later on this paper), i.e. they can be distributed across various machines in a network. For instance, the zipcode database may be located on a different machine as the StaffTracker web application.

To deploy the StaffTracker system, a system administrator or developer must install the databases on one or more MySQL servers. Moreover, all the web services and web application front-end must be built from source code, packaged, and activated on one or more Apache Tomcat servers.

There are various reasons why a developer or system administrator wants to deploy components from Figure 1 on multiple machines, rather than a single machine. For instance, a system administrator may want to deploy the StaffTracker web application on a machine which is publically accessible on the internet, while the data, such as the zipcodes, must be restricted from public access since they are privacy-sensitive. Moreover, each database may need to be deployed on a seperate machine, since a single machine may not have enough disk space available to store all the data sets. Furthermore, multiple redundant instances of the same component may have to deployed in a network, to offer load-balancing or failover.

Because of all these constraints, the deployment process of a system such as the StaffTracker becomes very complex. This complexity increases for every additional machine on which components of the system are deployed. Existing research and solutions are able to make the deployment easier for specific components and environments, but cannot manage a hetereogeneous system such as the StaffTracker on multiple platforms *and* offer properties such as dependency completeness and atomic upgrades. Moreover, they also do not offer the ability to guarantee non-functional requirements, such as privacy.

## 3. Background

Disnix is a toolset used to perform distributed deployment tasks and is built on top of Nix [4, 5], a package manager with some distinct features compared to conventional package managers (e.g. RPM [8]) to make deployment safe and reliable, such as model-driven deployment, complete dependencies, atomic upgrades and rollbacks and a garbage collector.

Nix stores packages in isolation from each other in a directory called the *Nix store*. Every package has a special file name such as /nix/store/5am52fmn...-firefox-3.6.6, in which the first part 5am52fmn... is a cryptographic hash derived from all the inputs (e.g. libraries, compilers, build scripts) used to build the component. If a user decides to build the component with, say, a different compiler, this will result in a different hash and thus a different filename in the Nix store. This approach makes it safe to store multiple versions and variants next to each other, because no components share the same name.

Since every component is stored in isolation in the Nix store rather than a global directory such as /usr/lib, we have stricter guarantees that its dependencies are correct and complete. With

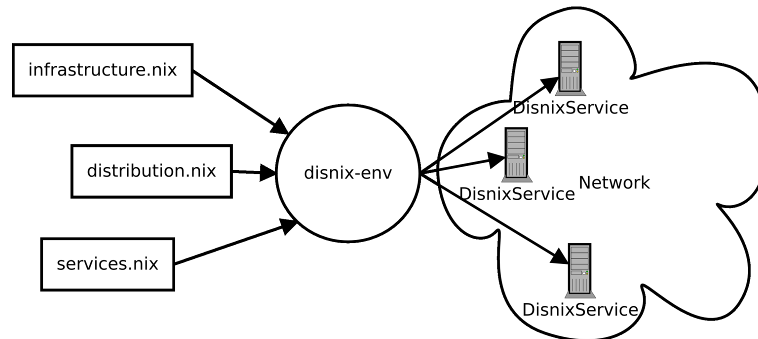



Figure 2: Overview of the disnix-env tool

conventional package managers the fact that a package builds successfully does not guarantee that the dependency specification is correct, since dependencies are stored in global locations. In Nix, all packages reside in the Nix store and must therefore be explicitly specified. This guarantees that if a package builds correctly on one machine, it will build on other machines of the same type as well.

Nix supports atomic upgrades and rollbacks, because components are stored safely next to each other and are never overwritten or automatically removed. Thus there is no time window in which a package has some files from the old version and some files from the new version (which would be bad because a program may crash if it is started during that period).

Moreover, Nix uses a purely functional domain-specific language called the Nix expression language to specify build actions. This makes building a package deterministic and reproducable. A garbage collector is included to safely remove packages that are no longer in use.

Because of these features, Nix is a very good basis to build a distributed deployment tool to make the deployment process of a distributed system, such as the StaffTracker efficient, reliable and atomic. However, Nix only deals with *intra-dependencies*, which are either run-time or build-time dependencies residing on the same machine. In order to deploy a distributed system into a network of machines additional features are provided by Disnix. Most importantly, these include models to describe machines in the network, and management of *inter-dependencies* – the run-time dependencies between components residing on different machines.

## 4. Overview

Figure 2 shows an overview of the disnix-env tool, which is used to perform the complete deployment process of a distributed system automatically from models. Three types of models specified in the Nix expression language are required as input parameters to automate the deployment process, each capturing a specific concern. All the remote deployment operations are performed by the *DisnixService*, a service that exposes deployment operations through a custom RPC protocol and must be installed on every machine in the network. All the deployment operations are initiated from a single machine, called the *coordinator*.

A *services* model is used to specify the available distributable components, how to build them, their *inter-dependencies*, and their *types*. The latter determine how a service is to be activated or deactivated. An *infrastructure* model is used to specify what machines are available

```
{distribution, system}:

let pkgs = import ../top-level/all-packages.nix { inherit system; }; in [1]
rec {
### Databases
  rooms = { [2]
    name = "rooms";
    pkg = pkgs.rooms;
    dependsOn = {};
    type = "mysql-database";
  };
  ...
### Web services
  RoomService = {
    name = "RoomService"; [3]
    pkg = pkgs.RoomService; [4]
    dependsOn = { [5]
      inherit rooms;
    };
    type = "tomcat-webapplication"; [6]
  };
  ...
### Web applications
  StaffTracker = {
    name = "StaffTracker";
    pkg = pkgs.StaffTracker;
    dependsOn = {
      inherit GeolocationService RoomService StaffService ZipcodeService;
    };
    type = "tomcat-webapplication";
  };
}
```

Figure 3: Partial services.nix model for the StaffTracker

in the network, how they can be reached in order to perform remote deployment operations, and other relevant capabilities and properties. A *distribution* model is used to specify a mapping of services to machines in the network.

Figure 3 shows a partial service model for the StaffTracker system. This expression is a set of attributes in which every attribute (such as [2]) represents a component from the architecture described earlier in Figure 1. Every attribute is itself an attribute set defining the relevant properties of the service, such as the name of the service [3], a reference to the function that builds the service from source code [4], the inter-dependencies of the service [5] (which correspond to the arrows shown in Figure 1) and its type [6]. The functions that build every component from source code and its intra-dependencies (such as libraries and compilers) are defined in a seperate file, which is imported at [1] (not covered in this paper).

Figure 4 shows an infrastructure model used to capture the machines in the network and their relevant properties/capabilities. This model is also an attribute set. Here, every attribute represents a system in the network, such as a machine called test2 [7]. For each machine relevant properties are defined such as the architecture of the system [10], so that Disnix can build a service for the desired type of platform, and a hostname attribute [8] so that Disnix knows how to reach a target machine to perform remote deployment operations. The other attributes are optional and are used to activate particular types of services. For instance, the tomcatPort attribute [9] specifies through which port the Apache Tomcat server can be reached.

A distribution model, shown in Figure 5, maps services to machines in the network. In this

```
{
  test1 = {
    hostname = "test1.testdomain.net";
    tomcatPort = 8080;
    system = "i686-linux";
  };

  test2 = { [7]
    hostname = "test2.testdomain.net"; [8]
    tomcatPort = 8080; [9]
    mysqlPort = 3306;
    mysqlUsername = "user";
    mysqlPassword = "secret";
    system = "x86_64-linux"; [10]
  };
}
```

Figure 4: An infrastructure.nix model for the StaffTracker

```
{infrastructure}:

{
  GeolocationService = [ infrastructure.test1 ]; [11]
  RoomService = [ infrastructure.test2 ];
  StaffService = [ infrastructure.test1 ];
  StaffTracker = [ infrastructure.test2 ];
  ZipcodeService = [ infrastructure.test1 infrastructure.test2 ]; [12]
  rooms = [ infrastructure.test2 ];
  staff = [ infrastructure.test2 ];
  zipcodes = [ infrastructure.test2 ];
}
```

Figure 5: A distribution.nix model for the StaffTracker

model, the name of each attribute corresponds to a service defined in the services model, while its value is a list of machines defined in the infrastructure model. For instance, [11] specifies that the GeolocationService must be deployed on the test1 machine defined in the infrastructure model. It is also possible to deploy multiple redundant instances of the same service by specifying multiple machines in a list, such as [12], which deploys the ZipcodeService on both test1 and test2. More details on how these models can be specified can be found in [9].

By writing instances of the three models mentioned above the user can deploy a complete system in a network of machines by running:

```
$ disnix-env -s services.nix -i infrastructure.nix -d distribution.nix
```

This command performs all the deployment steps to make the system available for use; i.e. *building* all the services defined in the services model from source code, *transferring* the services and all their intra-dependencies to the machines in the network and finally *activating* them in the right order derived from the dependency graph. By adapting the model instances above and by running disnix-env again, an *upgrade* is performed instead of a full installation. In this phase only the changed components are built from source code and transferred to the target machines and only obsolete services are deactivated and new services activated in the right order of activation. In case of a failure, Disnix will perform a *rollback*, which will bring the system back in its previous deployment state.

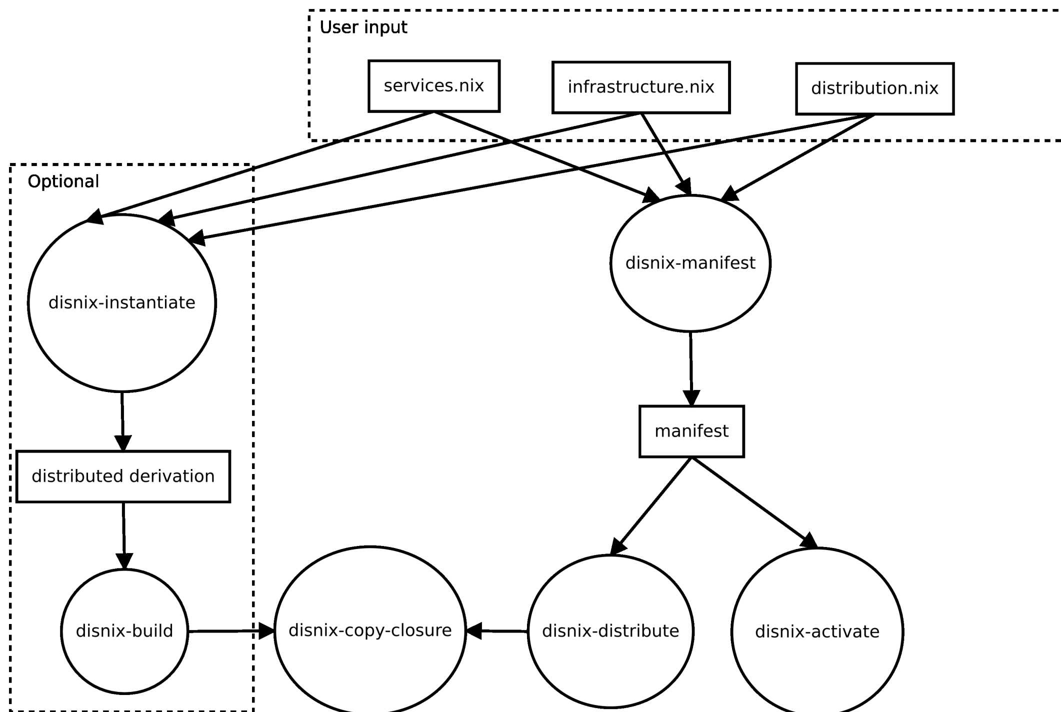



Figure 6: Architecture of disnix-env. Rectangles represent objects that are manipulated by the tools. The ovals represent a tool from the toolset performing a step in the deployment process.

A machine in the network may be of a different architecture than the machine distributing the services and thus incapable of building a service for that particular platform. Therefore, Disnix also provides the option to build services on the target machines in the network, instead of the coordinator machine.

Disnix has several important features. Since it is built on top of the Nix package manager, it will inherit properties to ensure that all intra-dependencies are always complete, components can be stored safely next to each other, and because Nix never overwrites files, an upgrade will not break the system in case of a failure. Moreover, Disnix can also be used to block or queue connections during the upgrade phase so that users will not be able to observe that the system is changing to make the upgrade process fully atomic. More details on this can be found in [9, 10].

## 5. Architecture

The disnix-env tool, which performs the complete deployment process, is composed of several tools each performing a step in the deployment process, shown in Figure 6.

Firstly, the disnix-manifest tool is invoked with the three models as input parameters. This tool generates a *manifest* file, an XML file that specifies what services need to be distributed to which machine and a specification of the services to be activated. The manifest file is basically a concrete version of the abstract distribution model. While the distribution model specifies

which services should be distributed to which machine, it does not reference the actual service that is built from source code for the given target machine under the given conditions, i.e. the component in the Nix store. This tool iterates over the distribution model and builds the services and all their dependencies for the given target machine and produces the manifest file referring to the build results.

The generated manifest file is then used by the disnix-distribute tool. This tool will efficiently copy the components and their intra-dependencies to the machines in the network through the remote interfaces. Only the components that are not yet distributed to the target machines are copied, while keeping components that are already there intact. Moreover, this process is also non-destructive, as no existing files are overwritten or removed. To perform the actual copy step to a single target, the disnix-copy-closure tool is invoked.

Finally, the disnix-activate tool is executed, which performs the *activation* phase. In this phase all the obsolete services are deactivated and all the new services are activated in the right order, by comparing the manifests of the current deployment state and the new deployment state. During this phase every service receives a lock and unlock request to which they can react, so that optionally a service can reject an upgrade when it is not safe or temporarily queue connections so that end users cannot observe that the system is changing. This allows the upgrade process to be completely atomic. More details about this process can be found in [9, 10].

Disnix provides an optional feature to build services on the target machines instead of the coordinator machine. If this option is enabled two additional modules are used. The disnix-instantiate tool will generate a distributed derivation file. This file is also an XML file similar to the manifest file, except that it maps store derivation files (low-level specifications that Nix uses to build components from source code) to target machines in the network.

It then invokes disnix-build with the derivation file as argument. This tool transfer the store derivation files to the target machines in the network, then builds the components on the target machines from these derivation files, and finally copies the results back to the coordinator machine. The disnix-copy-closure tool is invoked to copy the store derivations to a target machine and to copy the build result back.

Finally, it performs the same steps as it would do without this option enabled. In this case disnix-instantiate will not build every service on the coordinator machine, since the builds have already been performed on the machines in the network. Also, due to the fact that Nix uses a purely functional deployment model, a build function always gives the same result if the input parameters are the same, regardless of the machine that performed the build.

The disnix-env process is a composition of several other processes, for the following reasons:

- A user should be able to perform steps in the deployment process separately, for instance a user may want to build all the services defined in the model to see whether everything compiles correctly, while he does not directly want to activate them in the network.
- Testing and debugging is relatively easy, since everything can be invoked from the command line by the end user. Moreover, components can be invoked from scripts, to easily extend the architecture.
- Components are implemented with different technologies. The disnix-manifest and disnix-instantiate tools are implemented in the Nix expression language, because it is required to build components by the Nix package manager. Other tools are implemented in the C programming language such as the disnix-distribute and disnix-activate tools and shell scripts: disnix-env.

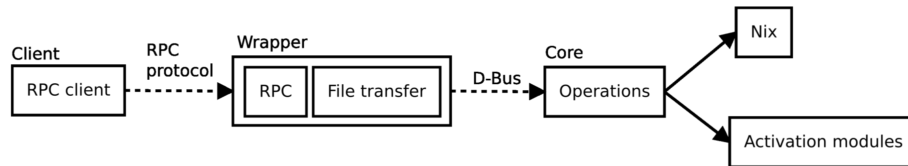



Figure 7: Architecture of the *DisnixService*

- Prototyping is relatively easy. A tool can be implemented in a high level language first and later reimplemented in a lower level language.

The principles above are quite common for developing UNIX applications and inspired by [11].

*5.1. Disnix service*

Some of the tools explained in the previous subsection have to connect to the target machines in the network to perform deployment steps remotely, e.g. to perform a remote build or to activate/deactivate a service. Remote access is provided by the *Disnix Service*, which must be installed on every target machine.

The Disnix service consists of two major components, shown in Figure 7. We have a component called the *core* that actually executes the deployment operations by invoking the Nix package manager to build and store components and the activation modules package to activate or deactivate a service. The *wrapper* exposes the methods remotely through an RPC protocol of choice, such as an SSH connection or through the SOAP protocol provided by an external add-on package called DisnixWebService.

There are two reasons why we choose to make a separation between the interface component and the core component:

- *Integration*. Many users have specific reasons why they choose a particular protocol e.g. due to organisation policies. Moreover, not every protocol may be supported on every type of machine.
- *Privilege separation*. To execute Nix deployment operations, super-user privileges on UNIX-based systems are required. The wrapper may need to be deployed on an application server hosting other applications, which could introduce a security risk if it is running as super user.

Custom protocol wrappers can be trivially implemented in various programming languages. As illustrated in Figure 7, the Disnix core service can be reached through the D-Bus protocol [12], used on many Linux systems for inter-process communication. In order to create a custom wrapper, a developer has to map RPC calls to D-Bus calls and implement file transport of the components. D-Bus client libraries exist for many programming languages and platforms, including C, C++, Java and .NET and are supported on many flavours of UNIX and Windows.

### 5.2. Disnix interface

In order to connect to a machine in the network, an external process is invoked by the tools on the deployment coordinator machine. An interface can be configured by various means, such as specifying the DISNIX_CLIENT_INTERFACE environment variable. The Disnix toolset includes two clients: disnix-client, a loopback interface that directly connects to the core service through D-Bus; and disnix-ssh-client, which connects to a remote machine by using a SSH connection. The separate DisnixWebService package includes an additional interface called disnix-soap-client, which uses the SOAP protocol to invoke remote operations and performs file transfers by using MTOM, an XML parser extension to include binary attachments without overhead.

A custom interface can be trivially implemented by writing a process that conforms to a standard interface, calling the custom RPC wrapper of the Disnix service.

### 5.3. Activation modules

Since services can represent many things, such as processes or databases, and cannot be activated generically, Disnix provides *activation types*, which can be connected to activation modules as illustrated in Figure 7. In essence, an activation module is a process that takes two arguments: the first is either 'activate' or 'deactivate' and the second is the Nix store path of the service that has to be activated/deactivated. Disnix passes all the properties defined in the infrastructure model as environment variables, so that system properties such as port numbers can be used.

The Disnix toolset includes a set of activation modules that can be used for activating commonly found services such as Apache Tomcat web applications, Apache HTTP server web applications, MySQL databases, NixOS configurations and generic UNIX processes. Moreover, there is a wrapper activation module, which will invoke a wrapper process with a standard interface included in the service.

Custom activation modules can be trivially implemented by a process that conforms to the activation module interface.

### 5.4. Libraries

Many of the tools in the Disnix toolset require access to information stored in models. Since this functionality is shared across several tools we implemented access to these models through libraries. The libmanifest component provides functions to access properties defined in the manifest XML file, libinfrastructure provides functions to access to properties defined in the infrastructure model and libdistderivation provides functions to access properties in a distributed derivation file.

We rather use library interfaces for these models, since they are only accessed by tools implemented in the C programming language and do not have to be invoked directly by end users.

### 5.5. Additional tools

Apart from disnix-env and the tools that compose it, the Disnix toolset includes several other utilities that may help a user in the deployment process of a system:

- disnix-query. Shows the installed services on every machine defined in the infrastructure.
- disnix-collect-garbage. Runs the garbage collector in parallel on every machine defined in the infrastructure model to safely remove components that are no longer in use.

- disnix-visualise. Generates a clustered graph from a manifest file to visualise services, their inter-dependencies and the machines to which they are distributed. The dot tool [13] can be used to convert the graph into an image, such as PNG or SVG.

- disnix-gendist-roundrobin. Generates a distribution model from a services and infrastructure model by applying the round robin scheduling method to divide services over each machine in the infrastructure model in equal portions and in circular order.

## 6. Extensions

Although the Disnix toolset itself provides useful features to make automated deployment of a distributed system possible and properties to make this process efficient and reliable, the toolset itself has a very generic approach. For instance, some non-functional properties cannot be addressed in a generic manner, since they are specific to the domain in which the system has to be used. We have to extend the toolset architecture to make this process more convenient, by integrating modules dealing with such non-functional properties.

In this section, we illustrate some of the extensions that we have designed. These extensions are implemented, but are still works in progress and not as mature and usable as the basic toolset.

### *6.1. Virtualization*

In some cases, especially while developing a system, users may want to test a certain deployment scenario in virtual machines, instead of performing a real deployment scenario, which requires having physical machines available somewhere with a preconfigured base system.

While Disnix manages the services constituting a distributed system, Disnix does not manage the system configurations of the target machines. This basically means that the virtual machines for testing have to be configured by other means (i.e. manually, by following the installation procedure of the operating system), which still could take a lot of effort.

In [14] we have demonstrated an approach in which we can automatically and efficiently instantiate a network of virtual machines from a declarative specification and automatically perform test cases on those virtual machines. This work builds upon NixOS [15], a GNU/Linux distribution which uses the Nix package manager to manage all packages, including the Linux kernel, system services and configuration files. Because Nix uses a purely functional approach for packages, we can share the Nix store components across virtual machines and the host system, which makes virtual machine instances relatively cheap.

To overcome the burden of creating virtual machines to test a deployment scenario with Disnix, we developed a tool called disnix-vm-env. This is a wrapper around the regular disnix-env that instantiates and launches virtual machines to simulate the network as defined by the models.

To do this, the infrastructure model is replaced by a *network model*, which defines the configuration of a network of NixOS machines. Figure 8 shows a partial network model, which can be used to deploy the StaffTracker in a virtual environment. This model is an attribute set in which every attribute represents a system, such as 13 representing machine test1. Each attribute refers to a NixOS configuration that describes the complete system configuration, such as the running services, configuration files and system properties. For instance at 14, the configuration of the Apache Tomcat container is specified, which is tuned with a specic Java virtual machine option to increase the heap space at 15. Other required services such as MySQL are defined at 16. This

```
{
  test1 = {pkgs, ...}: [13]
    {
      services.openssh.enable = true;
      services.disnix.enable = true;
      services.tomcat = { [14]
        enable = true;
        catalinaOpts = "-Xms64m -Xmx256m"; [15]
        ...
      };
      ...
    };
  test2 = {pkgs, ...}:
    {
      services.openssh.enable = true;
      services.disnix.enable = true;
      services.mysql.enable = true; [16]
      services.tomcat = {
        enable = true;
        catalinaOpts = "-Xms64m -Xmx256m";
        ...
      };
      ...
    };
}
```

Figure 8: Partial network.nix model for the StaffTracker

file can be used to automatically build a network of NixOS systems conforming to the given system configurations. More details about this model and its applications can be found in [14].

The main difference between the infrastructure model and the network model is that the infrastructure model is an *implicit* model; It should reflect the system configurations of the machines in the network, but the system administrator must make sure that the model matches the actual configuration. Moreover, it only has to capture attributes required for deploying the services. The network model is an *explicit* model, because it describes the *complete* configuration of a system and is used to deploy a complete system from this specification. If building a system configuration from this specifcation succeeds, we know that the specification matches the given configuration. Unfortunately, this property only works because we built upon NixOS; other operating systems such as FreeBSD or Windows cannot use this approach.

The following instruction from the command line spawns a network of machines defined in the network Nix expression and then deploys the StaffTracker system into the virtual machines:

```
$ disnix-vm-env -s services.nix -n network.nix -d distribution.nix
```

Figure 9 shows the architecture of the disnix-vm-env tool, which is a composition of several tools similar to the basic Disnix toolset. The disnix-generate-vms tool uses the network model to instantiate and launch a network of virtual machines running NixOS with the given configurations. The disnix-vm-generate-infra tool generates an infrastructure model from the network model which the disnix-env tool can use. Finally, the service model, generated infrastructure model and distribution model are used by disnix-env to perform the actual deployment on the virtual machines.

In order to connect to the *DisnixService* instances on the virtual machines a disnix-backdoor-client has been developed, which connects to each virtual machine through a UNIX domain

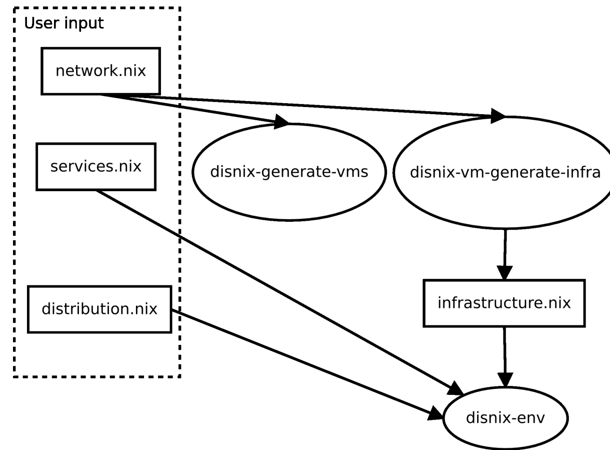



Figure 9: Architecture of disnix-vm-env

socket. This offers developers the advantage that no network settings have to be configured on the host system such as IP or port forwarding.

A limitation of this approach is that currently only NixOS configurations are supported. Deploying on a FreeBSD or Microsoft Windows system still requires the user to manually configure a base system.

*6.2. Dynamic deployment*

The deployment specifications required by Disnix are very *static*; a user must explicitly define all the machines and their relevant properties in the infrastructure model and has to explicitly define for each service to which machines they should be distributed. Maintaining such specifications by hand for large networks is impractical. Therefore a more dynamic approach is required. In [16] we have outlined the requirements for such an approach in which we want to *generate* an infrastructure and distribution model. Thus, we treat the infrastructure as a *dynamic cloud*, in which we automatically deploy components implementing a service to the right machines in the network, taking capabilities and quality of service attributes into account.

Although the Disnix toolset includes a very simple distribution generator called disnix-gendist-roundrobin, we would rather have a more sophisticated distribution generator that copes with non-functional requirements from the domain that we want to support.

By having a dynamic infrastructure and distribution generator, we can construct a service that continuously monitors the infrastructure and installed services and can react upon a change, e.g. perform a redeployment to fix the system if a machine breaks or to increase the capacity if a machine is added to the network.

Figure 10 shows an architecture of a dynamic deployment tool. The user provides a services model and a quality-of-service model, taking non-functional properties from the domain into account. A discovery service generates an infrastructure model. A distribution model generator maps the services to machines in the generated infrastructure model taking the constraints in the QoS model into account. Finally the services model, generated infrastructure and generated distribution model are passed to the disnix-env tool, which performs the actual deployment.

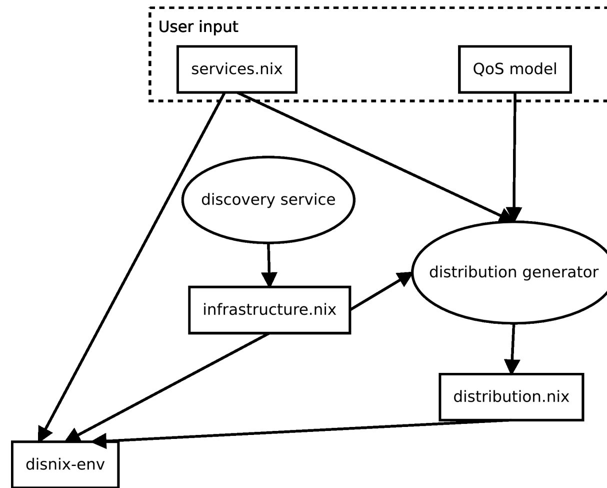



Figure 10: Architecture of a dynamic distributed deployment tool

Currently, we are working on a conceptual extension framework with an infrastructure model generator using Avahi [17] and a distribution model generator that uses a single variable (typically a system resource such as memory usage) to find a suitable distribution. Moreover, we are investigating non-functional requirements in the medical domain to build a more sophisticated dynamic deployment extension.

*6.3. Service design abstraction*

While Disnix provides properties to make the deployment of a distributed system safe and reliable and offers integration with modules to take non-functional requirements of the domain into account, not every non-functional property can be supported in the architecture of Section 6.2, because they are *implicit* in the design of a service.

An example of these issues are reliability issues, such as failing connections that are not automatically restored. If, for instance, the database connection with the rooms database fails and is not restored again, the RoomService will be inaccessible forever. Another possible issue is *static lookup* of inter-dependencies, which requires a user to deploy a new version of the service if a dependency is moved to another location. For example, if the zipcode database moves to another location all the inter-dependent services (ZipcodeService and the StaffTracker front-end) must be redeployed, since their configurations have to be adapted. Often service developers do not want to manually implement such constraints or even think about these issues, because they are too complex and too much effort.

We have developed ideas for higher level abstractions hiding such implementation details. One of our experiments is integrating WebDSL [18], a domain specific language for developing web applications with a rich data model into the dynamic deployment architecture.

By using an architecture integrated with a service abstraction layer (such as WebDSL for web applications), a developer only has to provide a high level specification of a web application and (optionally) some QoS attributes. The deployment architecture will make the resulting system available for use in a complex environment taking the QoS attributes into account.

## 7. Experience

### *7.1. Development*

Disnix has been under development for almost 2 years. The initial prototype was designed for use with SDS2, our first industrial case study, and was implemented by using the same technology, such as the Java programming language and the Apache Axis2 library to create a SOAP interface exposing deployment operations.

Later on, Disnix was applied to different types of distributed systems implemented with different technologies. Some initial implementation choices turned out to be impractical and therefore a new version of Disnix was developed.

Java was an impractical dependency, because it requires a Java Runtime Environment, which is quite large and not every platform has a decent Java Runtime implementation available. Moreover, it was impractical for systems not using Java, such as PHP web application systems, to deploy an entire Java environment to make deployment possible. Furthermore, it was also desirable for developers to execute certain tasks separately and to use different protocols than SOAP.

In the second version of Disnix all the Java modules were rewritten in C. This was not a very huge effort since their purposes were already well explored and the libraries used in the C version were almost as convenient as the Java ones. This also eliminated the large Java dependency, since the libraries used in the C versions were much smaller and quite common on Linux systems.

Moreover, a more modular architecture was designed in which communication was performed by an external process instead of using a Java interface. This choice gave us the option to directly invoke remote machine operations from the command-line and to use a programming language of choice to implement the interface (e.g. the disnix-soap-client was implemented in Java, since the Java SOAP libraries were more convenient to use and the disnix-ssh-client was implemented as a shell script not requiring a complex dependency such as Java).

Furthermore, the concept of activation types was developed in which types were mapped to external processes with a standard interface performing the activation and deactivation. Process composition was very practical in making the system more modular, since the modules can be implemented in various technologies and invoked directly by end-users.

In the third version (which became release 0.1) support for *heterogeneous* networks (networks consisting of machines with different architectures and operating systems) was implemented. The format of the models were changed in order to facilitate abstraction over architecture types.

### *7.2. Testing*

Disnix and its extensions are continuously built and tested by *Hydra*, our continuous build and integration server built around Nix [19]. Testing is done by using the approach described in [14] (also used for the virtualization extension described in Section 6.1), in which we declaratively instantiate cheap networks of virtual machines and perform automatic test-cases on them. By using this approach we continuously test the deployment operations of Disnix for various protocols.

### *7.3. Community*

Disnix is released as free and open source software under the GNU Lesser General Public License version 2.1 or higher. Extensions such as the activation modules and DisnixWebService are released under the MIT license.

Disnix is part of the *Nix project* (`http://nixos.org/`), in which solutions based around the Nix package manager are developed, such as NixOS, a GNU/Linux distribution using the Nix package manager and Hydra, our continuous build and integration server. All Nix related software is released under free and open source licenses.

The Nix project has a small community of developers and contributors consisting of researchers from various universities, developers from companies and enthusiastic individuals. Having a community offers us useful contributions such as compilers, libraries and end user applications in *Nixpkgs* [20], a repository consisting of over 2,500 packages that can be automatically deployed by the Nix package manager. These can be used to construct and deploy services with Disnix. Another major contribution is cross-compilation support which can also be used with Disnix. Maintaining all the parts in the Nix project ourselves is too much effort and would limit ourselves in achieving new results.

*7.4. Use cases*

We have used Disnix for automating the deployment of various types of distributed systems consisting of diverse kinds of services, such as MySQL databases, PHP web applications, Java based web applications, Java based web services and UNIX processes. Moreover, we have applied Disnix to an industrial case study from Philips Research called SDS2 [6], which is explained in [9].

Our repository contains several examples with full source code to demonstrate possible applications of Disnix. Some of these examples are:

- A PHP/MySQL web application system with multiple databases.
- The StaffTracker application described in this paper: a service-oriented system implemented mostly in Java, consisting of 3 layers (databases, web services, web application front-ends)
- Another Java-based service-oriented system to demonstrate how concepts such as load balancing and lookup can be implemented and used with Disnix.
- A distributed system consisting of processes communicating through TCP sockets, which can be adapted to block/queue connections to make the upgrade process atomic. This example is explained in [10].
- A network of NixOS configurations.

These examples can be deployed either on physical machines or tested in virtual machines by using the virtualization extension from Section 6.1. The examples are released under the MIT license.

## 8. Conclusion

In this paper we have given an overview of the Disnix toolset, which can be used to automatically deploy a distributed system into a network of machines. We have explained extensions on top of the basic toolset to deal with certain non-functional requirements from the domain that cannot be solved generically.

We have shown that we are using process composition to create a modular deployment architecture. This offers us various benefits, such as the ability to perform deployment steps separately, supporting tools implemented with various technologies, easy integration and easy prototyping. Moreover, we illustrated several extensions implemented on top of the Disnix toolset by using the process composition approach.

Furthermore, we have discussed our experiences developing Disnix, such as an open-source community that offers many benefits and the testing process in which we automatically instantiate a distributed test environment to test Disnix.

We have applied Disnix to various types of distributed systems consisting of various components, including an industrial case study in the healthcare domain designed by Philips Research.

Disnix, its extensions and examples as well as other Nix-related projects such as NixOS and Hydra are released as free and open source software, available from `http://nixos.org/`.

In the future we intend to maintain the Disnix toolset and further develop the extensions on top of the Disnix toolset. A dynamic deployment framework will be developed to support quality-of-services attributes in the medical domain. Moreover, a service abstraction model will be developed dealing with non-functional deployment requirements implicit in the design of a service. Furthermore, more evaluation is needed in large environments. Dynamic deployment support is a precondition to support this. Finally, another industrial case study is in progress, which is much larger than SDS2.

*Acknowledgements.* This research is supported by NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*. We wish to thank the contributors and developers of NixOS and SDS2, in particular Merijn de Jonge, who also contributed significantly to the development of Disnix.

## References

[1] A. Akkerman, A. Totok, V. Karamcheti, Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments, in: CD '05: Proc. of the 3rd Working Conf. on Component Deployment, Springer-Verlag, 2005, pp. 17–32.

[2] E. Caron, P. K. Chouhan, H. Dail, GoDIET: A Deployment Tool for Distributed Middleware on Grid 5000, Technical Report RR-5886, Laboratoire de l'Informatique du Parallélisme (LIP), 2006.

[3] R. S. Hall, D. Heimbigner, A. L. Wolf, A cooperative approach to support software deployment using the software dock, in: ICSE '99: Proc. of the 21st Intl. Conf. on Software Engineering, ACM, New York, NY, USA, 1999, pp. 174–183.

[4] E. Dolstra, E. Visser, M. de Jonge, Imposing a memory management discipline on software deployment, in: Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004), IEEE Computer Society, 2004, pp. 583–592.

[5] E. Dolstra, The Purely Functional Software Deployment Model, Ph.D. thesis, Faculty of Science, Utrecht University, The Netherlands, 2006.

[6] M. de Jonge, W. van der Linden, R. Willems, eServices for Hospital Equipment, in: B. Krämer, K.-J. Lin, P. Narasimhan (Eds.), 5th Intl. Conf. on Service-Oriented Computing (ICSOC 2007), pp. 391 – 397.

[7] MaxMind - GeoIP — IP Address Location Technology, `http://www.maxmind.com/app/ip-location`, 2010.

[8] E. Foster-Johnson, Red Hat RPM Guide, John Wiley & Sons, 2003.

[9] S. van der Burg, E. Dolstra, Automated deployment of a heterogeneous service-oriented system, in: I. Crnkovic, R. Mirandola (Eds.), 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society, 2010. To appear.

[10] S. van der Burg, E. Dolstra, M. de Jonge, Atomic upgrading of distributed systems, in: T. Dumitraş, D. Dig, I. Neamtiu (Eds.), First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp), ACM, 2008.

[11] E. S. Raymond, The Art of Unix Programming, Thyrsus Enterprises, 2003. Also available at: `http://www.faqs.org/docs/artu`.

[12] freedesktop.org - software/dbus, `http://www.freedesktop.org/wiki/Software/dbus`, 2010.

[13] Graphviz, `http://www.graphviz.org`, 2010.

[14] S. van der Burg, E. Dolstra, Automating system tests using declarative virtual machines, in: 21st IEEE International Symposium on Software Reliability Engineering (ISSRE 2010), IEEE Computer Society, 2010. To appear.
[15] E. Dolstra, A. Löh, NixOS: A purely functional Linux distribution, in: ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming, ACM, 2008.
[16] S. van der Burg, E. Dolstra, M. de Jonge, E. Visser, Software deployment in a dynamic cloud: From device to service orientation in a hospital environment, in: K. Bhattacharya, M. Bichler, S. Tai (Eds.), First ICSE 2009 Workshop on Software Engineering Challenges in Cloud Computing, IEEE Computer Society, 2009.
[17] Avahi, `http://avahi.org`, 2010.
[18] E. Visser, WebDSL: A case study in domain-specific language engineering, in: R. Lammel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Lecture Notes in Computer Science, Springer, 2008.
[19] E. Dolstra, E. Visser, The Nix Build Farm: A Declarative Approach to Continuous Integration, in: K. Mens, M. van den Brand, A. Kuhn, H. M. Kienle, R. Wuyts (Eds.), International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008).
[20] Nix Packages collection, `http://nixos.org/nixpkgs`, 2010.

# A Proof Repository for Formal Verification of Software

Michael Franssen

*Eindhoven University of Technology, Dept. of Mathematics and Computer Science, P.O. Box 513/HG5.36, 5600 MB Eindhoven, The Netherlands*

**Abstract**

We present a proof repository that provides a uniform theorem proving interface to virtually any first-order theorem prover. Instead of taking the greatest common divisor of features supported by the first-order theorem provers, the design allows us to support any extension of the logic that can be expressed in first-order logic. If a theorem prover has native support for such a logic, this is exploited. If the prover has no such support, the repository automatically uses the first-order encoding of the extension. A built-in proof assistant is provided that allows the user to manually guide the proving process when all provers fail to prove a theorem. To prove sub-theorems, the proof assistant is able to use the repository's full capabilities. The repository also maintains a database of proven theorems. When a requested theorem has been proved before, the result from the database is re-used instead of reconstructing the proof all over again. To test the repository, we constructed a tool for static verification of a basic programming language. This language is also described in this paper.

*Keywords:* Theorem proving, Program verification, Software tools

## 1. Introduction

Many tools for program verification use a layered model [1, 2]. The bottom layer exists of a logic, usually supported by some (semi)automatic tool. On top of this there is a layer of an intermediate language in which programs to be verified will be expressed. Typically, this intermediate language is simple, since it will not be used to directly write programs, but only as a stepping stone to verify programs in a more complex language. The programs in the intermediate language are explicitly annotated with assertions that express the program properties to be verified. For annotated programs written in the intermediate language a verification condition generator will calculate a set of logical conditions that must hold in order for the program to be correct. The upper layer consists of the actual programming language. Programs written in this language will first be translated into the intermediate language. During this translation, many complex features of the input language are expressed in the much simpler features of the intermediate language. Also, annotations are inserted into the intermediate program that claim

*Email address:* m.franssen@tue.nl (Michael Franssen)

absence of null-dereferences, array-index-out-of-bounds, etc. If the actual program is annotated with additional specifications, these specifications are translated into assertions too.

In order to obtain a high degree of automation, most tools select a single theorem prover to construct the required proofs and then tweak their proof obligations towards this theorem prover (ESCJava and SpecSharp use Simplify [1] and Perfect developer [3] has its own built-in theorem prover). When a proof fails, one can only add assumptions or tweak the program to get different verification conditions, hoping that these can be proved. Also, none of the aforementioned tools keep track of the constructed proof. So even if only a small part of the code changed, all the verification conditions are sent to the prover again. Moreover, during the development of a program, the context of definitions and assumptions grows larger. Proofs of theorems that were already proved when this context was small will take much more time to reconstruct in a larger context.

In this paper, we present a proof repository that provides the following services: (1) A rich logic to denote program specifications, definitions and abstract datatypes. (2) A single interface to connect to a wide range of automated theorem provers. This way, the programming tool is not restricted to using just one. (3) An interactive proof assistant that allows the user to guide the proof by manually constructing it himself as much as necessary. Instead of guessing assertions that might help the prover, this allows the user to gain insight in *why* a proof fails and what lemmas are needed to complete the proof. (4) A database of completed proofs that avoids reconstructing proofs for completed theorems all over again. This also implies that the allotted time to prove the verification conditions can be spent entirely on new or altered verification conditions.

To test the repository, we also introduce a basic programming language designed for modular verification. This programming language is a small extension of the guarded command language which is used, among others, by ESC/Java as an intermediate language to express programs to be verified. Nevertheless, a number of tree and list algorithms can be conveniently expressed in this language, since it exploits the repository's inductive type definitions.

## 2. Tool Structure

The structure of the entire toolkit described in this paper is depicted in Figure 1. The user enters a program in the text editor. This program is read by a JavaCC generated parser. The result is type-checked and then verification conditions are computed (see Section 4). Proofs for the verification conditions are obtained from the proof repository.

The repository first checks for every request whether or not the theorem exists in the database. If so, the stored result is returned. If not, one or more external theorem provers are launched (either remotely or on the same machine). For each different prover, the theorem is first translated into the native input language of that prover by a connector object. The order and time limits for the theorem provers can be configured by the user. If a proof is found, it is stored in the database for future re-use and sent back to the programming tool. If no proof is found, the proof assistant is launched to let the user manually construct a proof. The proof assistant lets the user select tactics that invoke small proof-steps. At any point the user may request a proof for a sub-theorem from the repository. In practice the proof-assistance is mainly used to perform a few initial steps and indicate where induction should be used to complete a proof. Automated theorem provers typically cannot handle proofs that require induction.

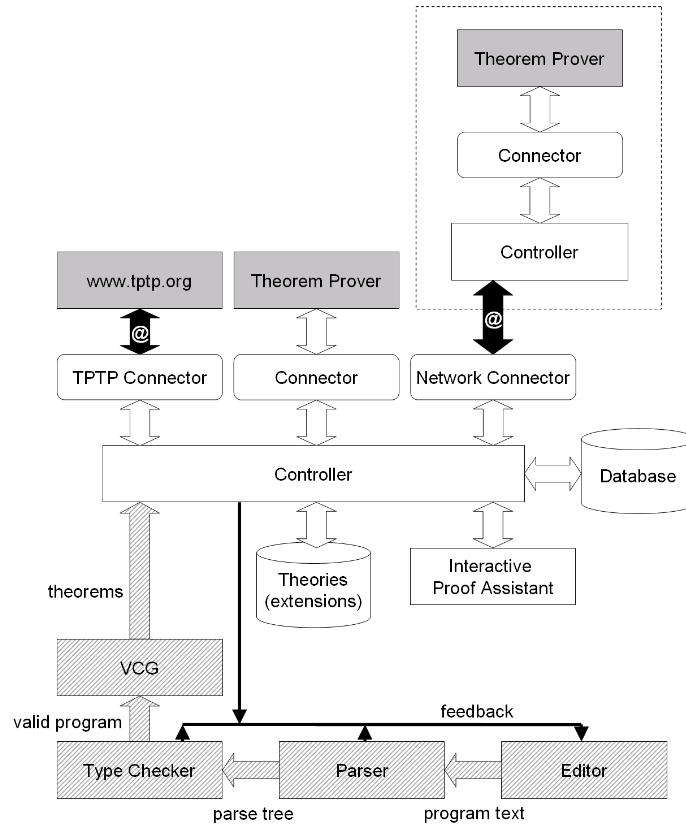



Figure 1: Architecture of the toolkit: Grey boxes represent external systems. White boxes are part of the repository. Textured boxes are part of the programming toolkit. Arrows with a @ sign represent internet connections.

Although the main contribution of our tool is the proof repository with support for inductive types and recursive definitions, we start in Section 3 by describing the programming language used to test the repository. This also introduces the notation for formulas and inductive definitions used by the repository. In Section 4, we explain how to compute the verification conditions. This demonstrates the kind of theorems for which the repository was constructed. We then describe the modules that make up the proof repository in Section 5. In Section 6 we explain how we handle inductive types and recursive function definitions. We conclude in Section 7.

## 3. The Programming Language

The programming language employed by our tool is an example language to show the usefulness of the repository. It is designed to be able to express some basic programming problems and to support simple generation of verification conditions.

The language is defined by the grammar in Figure 2. It is based on the guarded command language of Dijkstra, since he first proposed the weakest precondition calculus to establish formal correctness of programs. As a result, we support non-deterministic guarded **if** and **do** statements and multiple assignments. In order to make the language a bit more powerful and a more realistic test case for the proof repository, we added procedures, inductive datatypes and recursive specification functions. We will first discuss the main features of this language informally. In Section 4 we discuss how the tool computes the verification conditions.

| | | |
|---|---|---|
| *Prog* | ::= | $\textbf{program}\ \mathit{Id};\ \mathit{Decl}{*}\ \textbf{var}\ \mathit{Idlt}\ \{\textbf{post:}\ \mathit{Pred}\}\ \mathit{Stat}$ |
| *Decl* | ::= | $\textbf{procedure}\ \mathit{Id}(\mathit{Idlt}, \textbf{var}\ \mathit{Idlt})\ \{\textbf{pre:}\ \mathit{Exp}\ \textbf{post:}\ \mathit{Exp}\ [\Downarrow]\}\ \mathit{Stat}$ |
| | | $\mid \textbf{type}\ \mathit{Id}\ \textbf{is}\ \mathit{Id}(\mathit{Idlt})\ [\Box\ \mathit{Id}(\mathit{Idlt})]*\ \textbf{end}$ |
| | | $\mid \{\textbf{define}\ \mathit{Id}(\mathit{Idlt}) : \mathit{Id}\ \textbf{as}\ (\mathit{Match} \mid \mathit{Exp})\ \textbf{end}\}$ |
| *Stat* | ::= | $\mathit{Stat}; \mathit{Stat} \mid \textbf{skip} \mid \mathit{Idl}[, \mathit{Idl}]* := \mathit{Exp}[, \mathit{Exp}]* \mid \lvert[\textbf{var}\ \mathit{Idlt}; \mathit{Stat}]\rvert$ |
| | | $\mid \textbf{if}\ \mathit{GStats}\ \textbf{fi} \mid \{\textbf{inv:}\ \mathit{Exp}\ [\textbf{dec:}\ \mathit{Exp}]\}\textbf{do}\ \mathit{GStats}\ \textbf{od}$ |
| | | $\mid \textbf{match}\ \mathit{Exp}\ \textbf{with}\ \mathit{GStats}\ \textbf{end}$ |
| | | $\mid \mathit{Id}(\mathit{Expl})$ |
| *GStats* | ::= | $\mathit{GStats}\ \Box\ \mathit{GStats} \mid \mathit{Exp} \rightarrow \mathit{Stat}$ |
| *Match* | ::= | $\textbf{match}\ \mathit{Id}\ \textbf{with}\ \mathit{Id}(\mathit{Idl}) \rightarrow \mathit{Match}[\Box \mathit{Id}(\mathit{Idl}) \rightarrow \mathit{Match}]*\ \textbf{end}$ |
| *Exp* | ::= | $\mathit{Exp} = \mathit{Exp} \mid (\forall \mathit{Id} : \mathit{Id}.\mathit{Exp}) \mid (\exists \mathit{Id} : \mathit{Id}.\mathit{Exp}) \mid \mathit{Id}[(\mathit{Expl})]$ |
| | | $\mid \mathit{Exp} \wedge \mathit{Exp} \mid \mathit{Exp} \vee \mathit{Exp} \mid \neg \mathit{Exp}$ |
| | | $\mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} - \mathit{Exp} \mid \mathit{Exp} * \mathit{Exp}$ |
| *Expl* | ::= | $[\mathit{Exp}[, \mathit{Exp}]*]$ |
| *Id* | ::= | $\langle \mathit{identifier} \rangle$ |
| *Idl* | ::= | $[\mathit{Id}[, \mathit{Id}]*]$ |
| *Idlt* | ::= | $[\mathit{Id} : \mathit{Id}[, \mathit{Id} : \mathit{Id}]*]$ |

Figure 2: The grammar for the example programming language

### *3.1. Scope rules*

Basically, there are two contexts: The program context and the specification context. The program context contains all the variables and definitions that are available to program statements. These are written outside the curly brackets. The specification context contains additional definitions that are available to program specifications (pre and post conditions, invariants, function definitions, etc.). In specifications, all definitions and variables of the program context and the specification context can be used.

Parameters of definitions and procedures only exist within the body of the definition or procedure. Local variables can be declared between |[ and ]|. These variables only exist within this range. Local variable blocks can be nested. Within procedures, only variables in the parameter list and locally declared variables exist. Global variables are not supported in order to avoid aliasing problems.

The main program can only alter the variables introduced in its own **var** part and locally declared variables.

*3.2. Inductive Types*

Our language supports inductive type definitions called strictly positive types (See [4]). Given the syntactic restrictions of our language, all types that can be denoted are strictly positive and hence, well defined. Hence, there is no need to discuss the theory of strictly positive types in this paper. Variables of inductive types have value semantics (sometimes called copy semantics).

There is no need for built-in types, but the language assumes fixed definitions of *bool* and *int*. These types are additionally supported by basic operators and universal and existential quantifiers ∀ and ∃. Moreover, guards of **if** and **while** statements must be of type *bool* and may not contain any quantifiers. Also, we have the polymorphic boolean relation = between elements of any datatype.

All datatypes other than *bool* and *int* must be defined within the program itself. For example, lists of integers can be defined by:

$$\textbf{type}\ \mathit{list}\ \textbf{is}\ \mathit{empty}()\ \Box\ \mathit{cons}(x : \mathit{int}, \mathit{tail} : \mathit{list})\ \textbf{end}$$

*3.3. Specification Functions*

Functions can be defined only within the specification context, since the type of definitions allowed does not guarantee computability of the function. Once a function is defined, a procedure can be written that actually computes it. When this procedure is proved correct, the function apparently was computable. For instance, the function to compute the length of a list of integers is written as:

$$\begin{array}{l}
\{\textbf{define}\ \mathit{length}(L : \mathit{list}) : \mathit{int}\ \textbf{as} \\
\quad \textbf{match}\ L\ \textbf{with} \\
\quad\quad \begin{array}{lll} & \mathit{empty}() & \rightarrow\ 0 \\ \Box & \mathit{cons}(x, \mathit{tail}) & \rightarrow\ \mathit{length}(\mathit{tail}) + 1 \end{array} \\
\quad \textbf{end} \\
\ \textbf{end}\}
\end{array}$$

Since function definitions can only occur in the declaration part of a program, they can never refer to any program variables. Therefore, the match-pattern (following the keyword **match**) must be a variable referring to one of the function parameters. Following the **with** keyword, all constructors of the recursive type of this parameter are listed (including dummies representing the constructor's parameters) followed by → and a result-expression. This result-expression can be a direct expression or a match-expression. Note that by definition of the grammar, match-expressions are no ordinary expressions, since they cannot occur as sub-expressions at arbitrary places. They are only allowed when providing specification function definitions.

*3.4. Match-Statements*

To write a procedure implementing a specification function, we support match-statements. Although they are similar to match-expressions there are important differences: (1) The match-pattern can be any expression instead of just a variable. (2) Result-expressions are replaced by statements (3) Match-statements are ordinary statements. Semantically, a match-statement is like an if-statement where all the guards have a specific form and introduce a set of local variables to the corresponding branch.

### 3.5. Procedures

Once a procedure $p(a_1, \ldots, a_n, \textbf{var } x_1, \ldots x_m)\{\textbf{pre: } P \textbf{ post: } Q\}\ S$ is defined, it can be called from all succeeding procedures and the main program. $S$ can only refer to its parameters and locally declared variables. In order to avoid aliasing problems, all arguments of variable parameters in a procedure call must be different. Also, expressions used for value parameters in a procedure call may not depend on any of the variables parameter arguments. We denote $p$'s precondition by $p$.**pre** and $p$'s postcondition by $p$.**post**.

### 3.6. Optional termination proofs

It is up to the programmer whether or not termination of (part) of the program has to be proved. If a bound expression is provided for a loop (after the keyword **dec**), the required termination verification conditions are generated. Since procedures cannot be recursive, it is sufficient to add the optional $\Downarrow$ to the postcondition. Terminating procedures may only call other terminating procedures and all loops within its body must terminate. Loops within terminating loops must also terminate. The tool checks whether the required bound expressions and $\Downarrow$ are present in these cases.

## 4. Verification Conditions

The tool parses a string according to the given grammar using a JavaCC generated parser. The resulting tree is type-checked to find out if the program is valid. If so, verification conditions are generated based on a weakest precondition approach. That is, for any statement and post-condition, we compute a weakest precondition that must hold in the initial state, to guarantee that the postcondition holds in any final state. Apart from the weakest precondition, a set of side-conditions are computed that also must hold in order for the program to be correct. Based on this weakest precondition, the verification conditions for procedures and the main program are computed.

Every verification condition generated is labeled with a name that clarifies the property it expresses. In the remainder of this paper, an overlined term represents a comma-separated list of terms of the appropriate length.

**Definition 1.** *Weakest Preconditions*
*The weakest precondition* $wp(S, Q)$ *for statement* $S$ *and postcondition* $Q$ *is defined as:*

$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$$

$$wp(\textbf{skip}, Q) = Q$$

$$wp(\overline{x} := \overline{e}, Q) = Q[\overline{x} := \overline{e}]$$

$$wp(|[\textbf{var}\overline{x : T}; S]|, Q) = (\forall \overline{x : T}. wp(S, Q))$$

$$\begin{array}{llll} wp( & \textbf{if} & G_1 \rightarrow S_1 & = G_1 \Rightarrow wp(S_1, Q) \\ & \square & \ldots & \quad \wedge \ldots \\ & \square & G_n \rightarrow S_n & \quad \wedge\ G_n \Rightarrow wp(S_n, Q) \\ & \textbf{fi} & , Q) & \quad \wedge\ (G_1 \vee \ldots \vee G_n) \end{array}$$

$$\begin{array}{lll} wp(\{\textbf{inv:}I\ \textbf{dec:}B\} & & = I \\ \quad \textbf{do} & G_1 \rightarrow S_1 & \\ \quad \square & \ldots & \\ \quad \square & G_n \rightarrow S_n & \\ \quad \textbf{od} & , Q) & \end{array}$$

*Also, the following side-conditions have to be proved:*

| | |
|---|---|
| *invariance:* | $I \wedge G_i \Rightarrow wp(S_i, I)$ *for any* $1 \leq i \leq n$ |
| *finalisation:* | $I \wedge \neg G_1 \wedge \ldots \wedge \neg G_n \Rightarrow Q$ |
| *boundness:* | $I \wedge G_i \Rightarrow 0 \leq B$ *for any* $1 \leq i \leq n$ |
| *progress:* | $I \wedge G_i \wedge B = C \Rightarrow wp(S_i, B < C)$ *for any* $1 \leq i \leq n$ *where* $C$ *is a fresh constant* |

*boundness and progress only have to be proved if the optional B is provided*

$$\begin{array}{ll} wp(\textbf{match}\ E\ \textbf{with} & = E = C_1(\overline{v_1}) \Rightarrow wp(S_1, Q) \\ \quad C_1(\overline{v_1}) \rightarrow S_1 & \quad \wedge \ldots \\ \quad \square\ \ldots & \quad \wedge\ E = C_n(\overline{v_n}) \Rightarrow wp(S_n, Q) \\ \quad \square\ C_n(\overline{v_n}) \rightarrow S_n & \\ \quad \textbf{end}, Q) & \end{array}$$

$$wp(p(\overline{e}, \overline{v}), Q) = p.\textbf{pre}[\overline{a}, \overline{x} := \overline{e}, \overline{v}]$$

*Also, the following side-condition has to be proved:*

*correct use of p:* $p.\textbf{post}[\overline{a}, \overline{x} := \overline{e}, \overline{v}] \Rightarrow Q$

*Note that in order to prove a side condition with free variables, we actually prove its universal closure.*

Using the *wp* function from Definition 1 directly would yield a very rigid system. A procedure's postcondition would always have to match the entire postcondition of a procedure call. Usually, a procedure is used to satisfy only part of the postcondition.

To relax the verification conditions, we will split a postcondition into two parts: A part that is altered by the program and the part that is independent of the program. In order to define this split, we first provide some definitions.

$FV(P)$ denotes the free variables in $P$ and is defined as usual.

**Definition 2.** *Altering Variables*
*Given a program $S$, the set of variables that can possibly be altered during execution of $S$ is computed by the function $AV : Stat \rightarrow \mathcal{P}(\mathcal{V})$. AV is defined as:*

$$
\begin{array}{lcl}
AV(S_1; S_2) & = & AV(S_1) \cup AV(S_2) \\
AV(\textbf{skip}) & = & \emptyset \\
AV(\overline{x} := \overline{e}) & = & \{\overline{x}\} \\
AV(|[\textbf{var}\ \overline{x} \bullet S]|) & = & AV(S) \setminus \{\overline{x}\} \\
AV(\textbf{if}\ G_1 \rightarrow S_1 \Box \ldots \Box\ G_n \rightarrow S_n\ \textbf{fi}) & = & (\bigcup_{1 \le i \le n} AV(S_i)) \\
AV(\{\textbf{inv:}\ I\}\textbf{do}\ G_1 \rightarrow S_1 \Box \ldots \Box\ G_n \rightarrow S_n\ \textbf{od}) & = & (\bigcup_{1 \le i \le n} AV(S_i)) \\
AV(p(\overline{e}, \overline{x})) & = & \{\overline{x}\}
\end{array}
$$

*Note that AV is computed from the syntax of the statement. Stated simply: $AV(S)$ lists all free variables that occur in $S$ either at the left-hand side of an assignment statement or as an argument for a var-parameter of a procedure call.*

**Definition 3.** *Keeps and Alters*
*Let $Q = Q_1 \wedge \ldots \wedge Q_n$ be a predicate and let $S$ be a statement. Without loss of generality, assume that $FV(Q_i) \cap AV(S) = \emptyset$ for $0 \le i < j$ and $FV(Q_i) \cap AV(S) \ne \emptyset$ for $j \le i < n$ for some $j$ (if needed, we can re-order the conjuncts). We define Keeps and Alters as follows:*

$$
\begin{array}{lcl}
Keeps(S, Q) & = & Q_1 \wedge \ldots \wedge Q_{j-1} \\
Alters(S, Q) & = & Q_j \wedge \ldots \wedge Q_n
\end{array}
$$

Since $Keeps(S, Q)$ does not depend on any variable that can be changed by $S$, it must hold in the precondition iff it has to hold in the postcondition. We will use this to change the computation of *wp* for loops and procedure calls (changing *wp* for other statements is not necessary).

**Definition 4.** *wp with Keeps and Alters*
*We use $S$ to denote the entire statement, $K$ to denote $Keeps(S, Q)$ and $A$ to denote $Alters(S, Q)$. wp is redefined for loops and procedure calls by:*

$$
\begin{array}{lcl}
wp(\{\textbf{inv:}I\ \textbf{dec:}B\} & = & K \wedge I \\
\quad \textbf{do}\ \ G_1 \rightarrow S_1 & & \\
\quad \Box\ \ \ldots & & \\
\quad \Box\ \ G_n \rightarrow S_n & & \\
\quad \textbf{od}\ \ , Q) & &
\end{array}
$$

*The following side-conditions have to be proved:*
*invariance:* $K \wedge I \wedge G_i \Rightarrow wp(S_i, I)$ *for any* $1 \le i \le n$
*finalisation:* $K \wedge I \wedge \neg G_1 \wedge \ldots \wedge \neg G_n \Rightarrow A$
*boundness:* $K \wedge I \wedge G_i \Rightarrow 0 \le B$ *for any* $1 \le i \le n$
*progress:* $K \wedge I \wedge G_i \wedge B = C \Rightarrow wp(S_i, B < C)$ *for any* $1 \le i \le n$
*where C is a fresh constant*
*boundness and progress only have to be proved if the optional B is provided*

$$wp(p(\overline{e}, \overline{v}), Q) \quad = \quad K \wedge p.\textbf{pre}[\overline{a}, \overline{x} := \overline{e}, \overline{v}]$$

*The following side-condition has to be proved:*
*correct use of p:* $K \wedge p.\textbf{post}[\overline{a}, \overline{x} := \overline{e}, \overline{v}] \Rightarrow Q$

Computing weakest preconditions is not sufficient to obtain all verification conditions for a program. Therefore, we will define the function *VC*, based on *wp* to compute verification conditions for procedures and the main program.

**Definition 5.** *Verification conditions*
*The function VC computes the verification conditions that must be proved in order to establish the correctness of an entire program. It is defined as:*

$$
\begin{array}{lcl}
VC(\ \textbf{program}\ n & = & VC(Proc) \\
\quad Proc & & \cup\ \{correctness\ of\ n{:}(\forall_{\overline{x}}.wp(S,Q))\} \\
\quad \textbf{var}\ \overline{x} & & \\
\quad \{\textbf{post:}\ Q\} & & \\
\quad |[S\,]|\ ) & &
\end{array}
$$

$$
\begin{array}{lcl}
VC(\ \textbf{procedure}\ p(\overline{a},\textbf{var}\ \overline{x}) & = & \{correctness\ of\ p{:}(\forall_{\overline{a},\overline{x}}.P \Rightarrow wp(S,Q))\} \\
\quad \{\textbf{pre:}\ P\ \textbf{post:}\ Q\} & & \\
\quad |[S\,]|\ ) & &
\end{array}
$$

Proving all side conditions obtained by computing *wp* and the conditions computed by *VC* establishes full correctness of the entire program.

**Example 1.** *power*
*By means of a hello world example, we give a program to efficiently compute $a^b$ on natural numbers. In this example, the power function is defined and implemented. Since recursive function definitions are only available in specifications, users are forced to call the procedure, which (in this case) is much more efficient.*

```
...
{define pow(a, b : nat) : nat as
  match b with
        0       → 1
    □   suc(x)→ a ∗ pow(a, x)
  end
end}

procedure power(A, B : nat, var p : nat)
{pre: true post: p = pow(A, B)  ⇓}
|[var a, b : nat;
  p, a, b := 1, A, B;
  {inv: p ∗ pow(a, b) = pow(A, B) dec: b}
  do b > 0  →
     if   even(b)   → a, b := a ∗ a, b div 2
     □    ¬even(b)→ p, b := p ∗ a, b − 1
     fi
  od
]|
...
```

9

*The generated verification conditions are:*

$$
\begin{array}{ll}
\textit{correctness of power:} & true \Rightarrow 1 * pow(A,B) = pow(A,B) \\
\textit{finalisation:} & p * pow(a,b) = pow(A,B) \wedge \neg(b > 0) \Rightarrow p = pow(A,B) \\
\textit{invariance:} & p * pow(a,b) = pow(A,B) \wedge b > 0 \Rightarrow \\
 & \quad even(b) \Rightarrow p * pow(a * a, b \; div \; 2) = pow(A,B) \\
 & \quad \wedge \neg(even(b)) \Rightarrow p * a * pow(a, b - 1) = pow(A,B) \\
 & \quad \wedge (even(b) \vee \neg(even(b))) \\
\textit{progress:} & p * pow(a,b) = pow(A,B) \wedge b > 0 \Rightarrow \\
 & \quad even(b) \Rightarrow b \; div \; 2 < b \wedge \neg(even(b)) \Rightarrow b - 1 < b \\
\textit{boundness:} & p * pow(a,b) = pow(A,B) \wedge \neg(b > 0) \Rightarrow 0 \leq b
\end{array}
$$

*For a long period of time in the Eindhoven Computer Science curriculum students were taught to manually derive programs from specifications. This involved (amongst other skills) constructing and proving verification conditions like the ones in this example. The proofs themselves are an order of magnitude larger than the program being derived. Hence, the chance of mistake in the manually constructed proofs was at least as big as the chance of a mistake in the constructed program.*

*Apart from the convenience of automatic proof construction, tool support also provides more certainty about the absence of errors in the proofs. In fact, for somewhat larger programs tool support is mandatory, since manually constructing the verification conditions becomes infeasible, let alone proving them.*

The verification conditions from the example can be sent directly to the proof repository. In order to solve them, induction will be needed.

## 5. A Modular Repository

The repository's task is to provide proofs for theorems on request. How it does this should not be the concern of the client application posing the requests. Hence, basically, the proof repository acts as a theorem prover. All verification conditions mentioned in Section 4 can be handled by our repository. Also, the inductive type definitions and recursive function definitions are passed as such.

### *5.1. The Architectural Modules*

The basic architectural design consists of a number of modules: A connector, which connects an external automated theorem prover to the system; A proof assistant, to interactively construct proofs; A database, which stores proofs that have already been constructed; and a controller, which connects all components in a configurable way to the user application.

The user application communicates with the controller and asks for the proof of a theorem. If the controller finds the theorem in the database, it is returned to the application. If not, one or more theorem provers are launched to construct a proof. A connector component is used to translate the theorem into the format of the theorem prover. If a proof is found, it will be stored in the

database and returned to the application. If no proof is found, the proof assistant can be used to manually construct one or the application is notified.

In the following subsections, we describe the components in more detail. We assume that a proof request from the user application has the form $\Gamma \vdash P$, where $\Gamma$ is the context of definitions and assumptions and $P$ is the theorem to be proved in this context.

### *5.1.1. Connector*

A connector is responsible for connecting an external theorem prover to the repository. In general it takes a request $\Gamma \vdash P$ and translates it into the form of the external prover. It then launches the prover to construct a proof and translates the output into the internal representation of the repository. We provide a separate connector for each prover, but all connectors provide the same interface.

Also, a connector solves the problem of slight differences in the logic supported by different provers. For instance, Simplify [5] offers internal support for arithmetic, like the operators $+$, $-$ and $*$, while Spass [6] and E [7] do not. We could neglect any extensions and only support first-order predicate logic with equalities, which is supported by all theorem provers considered. However, since built-in support for extensions is usually more efficiently than the corresponding axiomatizations, this would weaken our system.

Therefore, our repository supports any extension to the basic logic that can be defined by axiomatizations in first-order predicate logic. A connector recognizes the use of an extension and chooses to either use extensions of the external theorem prover or to provide the corresponding axiomatization. The axiomatization of a theory is given once in the internal representation of our repository and translated by a connector when needed.

As a result of this approach, our repository provides a very rich language to the user application. For example: One often needs lists, sets, stacks, etc. These can easily be axiomatized in first-order logic, but many basic theorems about them are also needed. Since not all theorem provers support these datatypes, one might be tempted to choose one specific prover at an early stage. By using the repository instead, one can use all extensions and connect to different provers as needed. Different provers have different specialties, but the user application can use them all.

Currently, the repository provides native support for the theorem provers Spass [8], Simplify [5], Z3 [9], and any theorem prover that accepts the TPTP [10] input format (like E-prover [7]). Also, through the TPTP webservice, more than 50 theorem provers (including different versions) are directly available. In case of Simplify and Z3, the repository uses their internal support for integer numbers instead of their translations to first-order axioms. This includes the operators $+$, $-$ and $*$.

How the translated theorem is sent to the prover depends on the interface provided by the external prover. Currently, a file is generated that contains the theorem in the external provers native format. The theorem prover is then launched as an external process on this file. The resulting output is parsed and included in the result. In case of the TPTP webservice, the file is sent over the internet to the TPTP servers and no external process is started.

### *5.1.2. Proof Assistant*

First-order logic is semi-decidable, meaning that although every valid theorem can (in theory) be proved automatically, an invalid theorem might cause an infinite proof-attempt. Hence, if after a given amount of time no proof has been found, one cannot distinguish between the theorem being incorrect or the amount of time being insufficient. A theorem prover therefore hardly ever gives other reasons than 'insufficient resources' (including time) if it fails to find a proof.

When the automatic theorem provers fail (for example, if induction is needed), it is necessary to (partially) construct a proof manually. For this, an interactive proof assistant is provided (an extension of the prover used in Cocktail [11]). In this proof assistant, single proof steps called tactics are used to split the main theorem in smaller sub-goals. These sub-goals can in turn be passed to the external automated theorem provers to complete the proof. If needed, the entire proof can be constructed manually by the interactive prover.

### *5.1.3. Theories*

The repository supports first-order logic with equalities, because it (1) is a well-known logic supported by many systems, (2) is at least semi-decidable and (3) is powerful enough to express meaningful theorems.

Several systems exist that have more efficient native support for theories often expressed in first-order logic (e.g. [5] has support for integer numbers and [12] is specialized in Clean programs). Since we want to exploit this special support, any theory that can be expressed in first-order logic is added as a theory module in the proof repository. A theory module consists of the first-order definitions and theorems needed to define this theory for external provers that do not natively support it. When a connector translates a theorem to an external prover, this theory module is added to the translation only if the external prover has no native support for the theory. Otherwise, the translation will exploit the native support, yielding better performance.

The theories module is not really a module as such. It is more like a library of translations from logic extensions to basic first-order logic that is used by connectors to deal with logic extensions that are not supported natively by the external theorem prover.

### *5.1.4. Database*

The repository provides a database in which proven theorems can be stored for re-use. Instead of storing each variant of a theorem, a normal form of the theorem is stored. When searching the database, we do not look for exactly the same theorem, but for a theorem which is "more general", i.e. a theorem that implies the theorem we seek. As a result, renaming program variables or adding assumptions does not invalidate previous results. Also, obviously equivalent formulas are considered equal (e.g. $P \Rightarrow Q$ and $\neg P \vee Q$). The normal form used and the notion "more general" is defined below.

We first define a Database Normal Form (DbNF) that puts all conjuncts, disjunct and universal quantifications together and eliminates implications and existential quantifications. This will allow searching the database in such a way that the exact order of these conjuncts, disjuncts and universal quantifications is irrelevant. The DbNF of a formula is computed as follows:

1. We start by computing the negation normal form of the formula, thereby eliminating $\Rightarrow$ and putting negations only in front of atomic formulas.
2. Next we replace all $\exists$-quantifications by skolem-functions. Equivalence of formulas is no then longer dependent on where exactly the $\exists$ formula is inserted. That is, $\exists x.(P(x) \wedge Q)$ and $(\exists x.P(x)) \wedge Q$ become equal because they both have the skolem form $P(s()) \wedge Q$, where $s$ is the skolem function replacing $x$. There are several ways to skolemize a formula. We use the rewrite rule $(\exists x.A) \rightarrow A[x := s(x_1, \ldots, x_n)]$, where $s$ is a fresh skolem function of arity $n$ and $\{x_1, \ldots x_n\} = FV(A) \setminus \{x\}$.
3. Then we turn the formula into head-normal form by putting all $\forall$ quantifiers in front of the formula.
4. Finally, we compute the conjunctive normal form of the unquantified parts.

We now have a formula of the form $\forall x_1, \ldots x_n.C_1 \wedge \ldots \wedge C_m$, where every $C_i$ is a disjunction $L_1 \vee \ldots \vee L_k$ of literals. The $C_i$ are called clauses.

We also define a Query Database Normal Form (QDbNF), which is only slightly different from DbNF. The QDbNF is needed, since the skolem functions in the database will be different from the ones in the query, yet they represent existentially quantified variables that might match. Therefore, in the QDbNF we keep the existential quantifiers, but skolemize universal quantifications.

The QDbNF of a formula $P$ is computed as follows: Perform steps 1 till 3 on the formula $\neg P$, yielding $P'$, which is a skolemized head-normal form in which negations only occur in front of atomic formulas. We then apply step 1 again on $\neg P'$ and finally apply step 4. The QDbNF has the form $\exists x_1, \ldots x_n.C_1 \wedge \ldots \wedge C_m$, where every $C_i$ is again a clause.

It is semi-decidable if a stronger theorem is already in the database. Therefore, we define a notion of "more general", which is a more restrictive, but computable claim about theorems. Instead of checking if a stronger theorem is already in the database, we will check if a more general theorem is in the database.

**Definition 6 (More general clause).**
*Let $L = L_1 \vee \ldots \vee L_n$ and $L' = L'_1 \vee \ldots \vee L'_m$ be clauses and let $\theta$ be a mapping from $FV(L) \cup FV(L')$ to ground terms. $L$ is said to be more general than $L'$ according to $\theta$ if there exists a mapping $\sigma : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ such that $\theta(L_i) = \theta(L'_{\sigma(i)})$ for $i = 1, \ldots, n$. Note that $\langle\rangle \vdash \theta(L) \Rightarrow \theta(L')$.*

**Definition 7 (More general CNF formula).**
*Let $P = P_1 \wedge \ldots \wedge P_n$ and $Q = Q_1 \wedge \ldots \wedge Q_m$ be formulas in conjunctive normal form (CNF) and let $\theta$ be a mapping from $FV(P, Q)$ to ground terms. $P$ is said to be more general than $Q$ if there exists a mapping $\sigma : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ such that $P_{\sigma(i)}$ is more general than $Q_i$ for $i = 1, \ldots m$. Note that $\langle\rangle \vdash \theta(P) \Rightarrow \theta(Q)$.*

**Definition 8 (More general (Q)DbNF formula).**
*Let $P$ and $Q$ be formulas, such that the DbNF of $P$ is $\forall x_1, \ldots x_n.P'$ and the QDbNF of $Q$ is $\exists y_1, \ldots y_m.Q'$. $P'$ and $Q'$ are in CNF. $P$ is said to be more general than $Q$ if there exists a mapping $\theta$ from $x_1, \ldots, x_n, y_1, \ldots y_m$ to ground terms such that $P'$ is more general than $Q'$.*

**Definition 9 (More general theorem).**
*A theorem $\Gamma \vdash P$ is defined to be "more general" than theorem $\Delta \vdash Q$ if $P$ is more general than $Q$ and for every formula in $\Gamma$ there is a more general formula in $\Delta$.*

**Theorem 1 (More general implies stronger).**
*If $\Gamma \vdash P$ is more general than $\Delta \vdash Q$, then $\Delta$ is stronger than $\Gamma$ and $P$ is stronger than $Q$, hence $\Gamma \vdash P$ is stronger than $\Delta \vdash Q$.*

We consider a theorem to be in the database if a more general theorem is in the database, which is a valid conclusion because of theorem 1.

In order to store a proven theorem $\Gamma \vdash P$ in the database, we first compute the QDbNF of all formulas in $\Gamma$ and the DbNF of $P$. This yields $\Gamma' \vdash P'$, which is actually stored.

When a query $\Delta \vdash Q$ is looked up in the database, we compute the DbNF of all formulas in $\Delta$ and the QDbNF of $Q$. The unquantified parts of the required normal forms of all formulas to find a more general theorem in the database are now directly available. The required substitutions are found by using a simple Robinson unification algorithm [13], which also solves the problem of the order of universal quantifications. By using an incremental, substitution free unification algorithm [14], the efficiency is comparable to syntactic comparison. To find the required mappings between conjuncts and disjuncts, we use trial and error (trying for each disjunct and conjunct every opposing disjunct and conjunct) yielding a quadratic search. This is made almost linear by imposing an ordering (like lexicographical path ordering, LPO) on the terms within the clauses.

*5.1.5. Controller*

The task of the controller is twofold: It provides an API to the user application to pose queries and it manages the tools it has available to answer them.

The queries posed by the user application must have the form $\Gamma \vdash P$, where $\Gamma$ is a list of declarations, definitions and assumptions called the context and where $P$ is a formula.

To answer the query, the controller first consults the database (if available). If the theorem is not in the database, external automated theorem provers are launched. The launching order and the timeout for each theorem prover can be configured within the controller. It is also possible to launch all theorem prover simultaneously and see which one find an answer to the query first.

The answer to a query can be *True*, meaning that $P$ holds in the given context $\Gamma$, *False*, meaning that $P$ does not hold in $\Gamma$, *unknown*, meaning that the repository processed the query completely, but cannot find an answer, and *error*, meaning that some error occurred while processing the query. The answer also contains other information, like the name and version of theorem prover that provided the result, the time taken to construct the proof and, if desired, the literal proof-output of the theorem prover.

Access to the repository is provided in two ways: (1) By a direct API of method calls of a Java object and (2) By commands issued through a socket stream to the repository (i.e. the repository acts as an internet service). While the first way is more direct and usually easier to implement for the user application, the second way opens a whole new perspective to use the repository. A special network connector object is provided that allows one repository to consult another

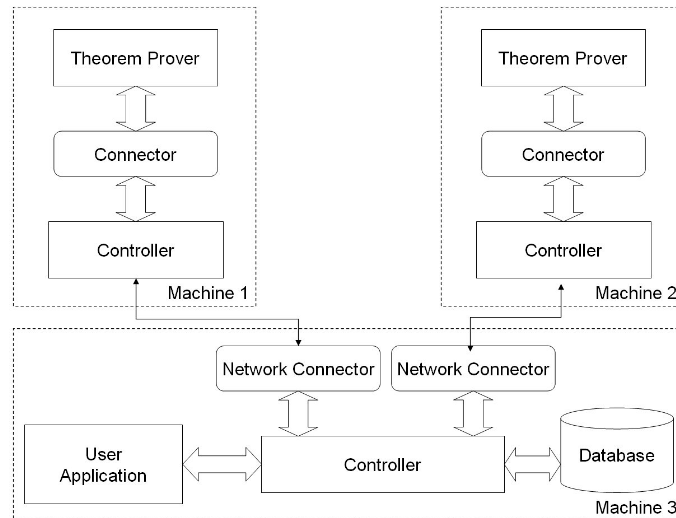



Figure 3: Using a separate machine for each theorem prover

repository as if it were a theorem prover. A controller then only configures the provers available on the local machine. By connecting the local repository to other repositories on other sites, the entire setup of the system becomes configurable. We will elaborate on this in Section 5.2.

### *5.2. Configurations*

The main user applications we had in mind when designing the repository were tools for proving correctness of programs. Usually these tools are linked to a specific prover. For instance, ESC/Java [1] and Boogie are linked to Simplify [5] and ZAP [15], the Omnibus system [16] is linked to PVS [17], Perfect Developer [3] and Cocktail [11] use their own internal provers, etc.

By connecting these systems to the repository instead, they are able to benefit from the strengths of several theorem provers. Also, proofs only have to be constructed once, since the database prevents the same proof from being constructed repeatedly. Reconstruction of proofs proved to be a drawback of systems that rely solely on automated theorem provers, like ESC/Java and Perfect Developer (see [18]).

Also, the modules can be used to boost computational power: One can configure one machine for each theorem prover, using only one connector and the controller (without database). The machine running the user application then consults the other machines through the special network connector instead of running its own theorem provers (see Figure 3). One can also setup one additional node that only runs the database to implement a central repository that is used by several user applications. An extreme case of this situation is to run the entire repository as Software as a Service.

Using the modules with the database only locally and with just one prover is already useful to

support reusing proofs. This will already speed up a user application by avoiding the needless reconstruction of proofs.

## 6. Handling inductive types and recursive functions

Inductive datatype definitions and recursive function definitions are supported by the repository and its built-in proof assistant. As useful as this kind of definitions are to the user, they are not part of standard first-order logic and not generally supported by first-order theorem provers. The reason for this is that proving properties of inductive type definitions usually requires induction proofs, but induction proofs only prove the property for elements of the inductive type with a finite size. Theorem provers do not assume that all models only map to values of finite size and hence, they do not use induction. We only consider programs about elements of finite size, so we can safely allow induction proofs in our system.

Since the theorem provers do not support inductive types, their definitions must be translated into a set of first-order formulas to send to the theorem provers.

We use a translation function *to1st* to translate inductive datatypes and recursive functions into first-order logic.

**Definition 10.** *Translation function to1st*
*Inductive type definitions and recursive specification function definitions are translated into sets of first-order declarations and axioms (assumptions) by the translation function to1st:*

$to1st(\textbf{type } t \textbf{ is } C_1(\overline{p_1}) \,\square \ldots \square\, C_n(\overline{p_n}) \textbf{ end}) =$
$\quad t : *_s;$
$\quad C_1(\overline{p_1}) : t;$
$\quad \ldots$
$\quad C_2(\overline{p_n}) : t;$
$\quad Cons_t : (\forall v : t.(\exists \overline{a_1}.t = C_1(a_1)) \vee \ldots \vee (\exists \overline{a_n}.t = C_n(a_n)));$

$*_s$ *is used to denote the set of all sets (i.e.* $t : *_s$ *declares that t is a type). The Cons_t axiom enables the external provers to make case distinctions when proving properties over elements of t (every element of t can be written as the result of one of its constructors). Note, that Cons_t can be derived from the inductive definitions by an induction proof.*

$to1st(\textbf{define } f(\overline{p}) : t \textbf{ as } M \textbf{ end}) =$
$\quad f(\overline{p}) : t;$
$\quad T(f(\overline{p}), id, M)$

*where id is the identity mapping (the empty substitution). T translates the match-expressions for a function to a set of assumptions.* $T(F, \theta, E)$ *with F a function,* $\theta$ *a substitution and E a (match-)expression is defined by:*

$T(F, \theta, \textbf{match } v \textbf{ with } C_1(\overline{a_1}) \rightarrow E_1 \,\Box \ldots \Box\, C_n(\overline{a_n}) \rightarrow E_n \textbf{ end}) =$
$T(F, [v \mapsto \theta(C_1(\overline{a_1}))] \circ \theta, E_1);$
$\ldots$
$T(F, [v \mapsto \theta(C_n(\overline{a_n}))] \circ \theta, E_n)$

$T(F, \theta, E) = C(\theta(F) = \theta(E))$ *if E is not a match-expression*

*where $C(P)$ denotes the universal closure $(\forall FV(P).P)$. These declarations and assumptions are such that any first-order theorem prover is able to handle them.*

**Example 2.** *trees For example, consider the following inductive definition of trees and the functions to compute the number of leafs and nodes in a tree:*

**type** *tree* **is** *leaf*(*x* : *int*) □ *node*(*L*, *R* : *tree*) **end**
{**define** *leafcount*(*t* : *tree*) : *int* **as**
  **match** *t* **with**
    *leaf*(*x*) → 1
  □ *node*(*L*, *R*)→ *leafcount*(*L*) + *leafcount*(*R*)
  **end**
**end**}
{**define** *nodecount*(*t* : *tree*) : *int* **as**
  **match** *t* **with**
    *leaf*(*x*) → 0
  □ *node*(*L*, *R*)→ *nodecount*(*L*) + *nodecount*(*R*) + 1
  **end**
**end**}

*The translation into first-order logic then becomes:*

$tree : *_s;$
$leaf(x : int) : tree;$
$node(L, R : tree) : tree;$
$Cons_tree : (\forall t : tree.(\exists x : int.t = leaf(x)) \vee (\exists L, R : tree.t = node(L, R)));$
$leafcount(t : tree) : int;$
$(\forall x : int.leafcount(leaf(x)) = 1);$
$(\forall L, R : tree.leafcount(node(L, R)) = leafcount(L) + leafcount(R);$
$nodecount(t : tree) : int;$
$(\forall x : int.nodecount(leaf(x)) = 0);$
$(\forall L, R : tree.nodecount(node(L, R)) = nodeount(L) + nodecount(R) + 1;$

*which are indeed first-order axiomatic definitions that can be handled by any theorem prover.*

*If we now request a proof for $(\forall t : tree.leafcount(t) = nodecount(t) + 1)$ from the repository, no theorem prover is able to proof it, by lack of induction (note that the property does not hold for infinite trees). We then indicate in the built-in proof assistant that we want to prove the main goal by induction (just selecting the correct tactic). This yields two new subgoals (see the screenshot*

*in Figure 4), namely the base and the induction cases:*

$$(\forall x : int.leafcount(leaf(x)) = nodecount(leaf(x)) + 1)$$
$$(\forall L, R : tree.(leafcount(L) = nodecount(L) + 1) \Rightarrow (leafcount(R) = nodecount(R) + 1) \Rightarrow (leafcount(node(L, R)) = nodecount(node(L, R)) + 1))$$

*Both of these can be proved by automated theorem provers connected to the repository.*

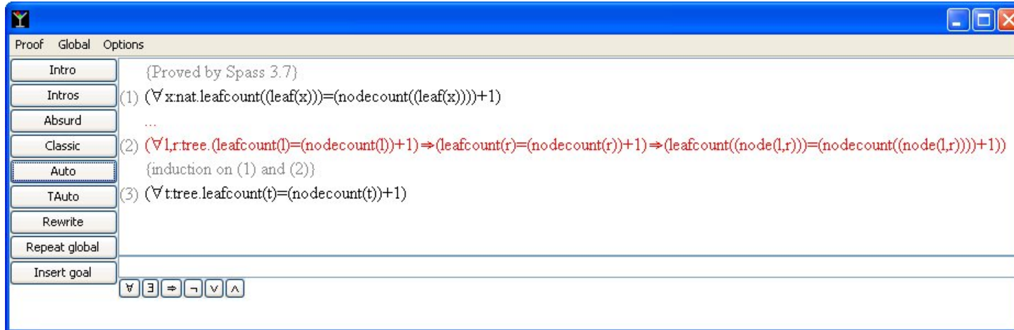



Figure 4: A screenshot of the proof assistant doing induction on trees.

## 7. Conclusions and Future Work

The language as described fulfils its purpose as a test suite for the proof repository. It can be used to express smaller algorithmic examples that lead to interesting verification conditions that cannot always be proved fully automatically.

The interactive theorem prover supports the inductive types and recursive definitions directly, allowing the user to split some of the harder theorems into simpler sub theorems that are then handled by the repository.

Since all proofs are stored in the database, verification conditions that have not changed after a small change in the program are not proved again. This constitutes a dramatic speed-up when re-verifying a program after small changes have been made. The time allotted to verify the program can now be entirely spent on those verification conditions that are actually new or changed. Also, since the database has some limited deductive capabilities, renaming variables or adding new definitions does not lead to re-constructing proofs of verification conditions.

Next, we want to extend the language to include more advanced features (recursive procedures, references, arrays, etc.) in order to write more interesting programs. Also, the language would then become more interesting to serve as an intermediate language to verify real-world programs.

We also will test the repository in several different situations (e.g. educational settings) and configurations (local repositories, a central database, etc) to find out what more is needed for large scale application.

## References


[1] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for Java, SIGPLAN Not. 37 (5) (2002) 234–245. doi:http://doi.acm.org/10.1145/543552.512558.

[2] M. Barnett, K. R. M. Leino, W. Schulte, Vol. 3362 of LNCS, Springer, 2005, Ch. The Spec♯ Programming System: An Overview, pp. 49–69.

[3] G. Carter, R. Monahan, J. Morris, Software refinement with Perfect Developer, Software Engineering and Formal Methods, IEEE, 2005, pp. 363–372.

[4] P. Morris, T. Altenkirch, N. Ghani, Constructing strictly positive families, in: CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2007, pp. 111–121.

[5] D. Detlefs, G. Nelson, J. B. Saxe, Simplify: A theorem prover for program checking, Journal of the ACM 52 (3) (2005) 365–473.

[6] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, SPASS Version 3.5, Automated Deduction–CADE-22 (2009) 140–145.

[7] S. Schulz, E – A Brainiac Theorem Prover, Journal of AI Communications 15 (2/3) (2002) 111–126.

[8] C. Weidenbach, B. Afshordel, E. Keen, C. Theobalt, D. Topić, Spass theorem prover, in: URL: `http://spass.mpi-sb.mpg.de/`, Max-Planck-Institut für Informatik, 2007.

[9] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, Tools and Algorithms for the Construction and Analysis of Systems (2008) 337–340.

[10] G. Sutcliffe, C. Suttner, The TPTP problem library, Journal of Automated Reasoning 21 (2) (1998) 177–203.

[11] M. Franssen, Cocktail: A tool for deriving correct programs, Ph.D. thesis, Eindhoven University of Technology (2000).

[12] M. de Mol, M. van Eekelen, A Proof Tool Dedicated to Clean, Applications of Graph Transformations with Industrial Relevance (2000) 254–257.

[13] J. Robinson, A machine oriented logic based on the resolution principle, Journal of the Association for Computing Machinery 12 (1) (1965) 23–41.

[14] M. Franssen, Implementing rigid E-unification, Computing Science Report 08–24, Eindhoven University of Technology (2008).

[15] T. Ball, S. Lahiri, M. Musuvathid, Zap: Automated theorem proving for software analysis, Tech. Rep. MSR-TR-2005-137, Microsoft Research (October 2005).

[16] T. Wilson, S. Maharaj, R. G. Clark, Flexible and configurable verification policies with omnibus, Journal on Software and Systems Modeling 7 (3) (2008) 257–272.

[17] S. Owre, J. Rushby, N. Shankar, PVS: A prototype verification system, in: D. Kapur (Ed.), 11th International Conference on Automated Deduction, Vol. 607 of Lecture Notes in Artificial Intelligence, CADE, Springer Verlag, 1992, pp. 748–752.

[18] H. Maassen, Verified design by contract : case studies, Master's thesis, Eindhoven University of Technology (2008).

# Processes and Practices for Quality Scientific Software Projects

Veit Hoffmann[a], Horst Lichter[a], Alexander Nyßen[b]

[a] *Research Group Software Construction, RWTH Aachen University, Aachen, Germany*
[b] *itemis AG, Lünen, Germany*

**Abstract**

Nowadays modern software development processes are well established and are one of the mayor success factors for software projects. However, many software projects in scientific organisations have serious quality issues since they lack a feasable development process. In this paper we discuss a feature based development process for scientific software projects that addresses the specific characteristics of scientific software. Moreover, we introduce a set of best practices for the infrastructure and management of such projects.

*Keywords:* software development process, best practice, scientific software

## 1. Introduction

Software tools are a vital prerequisite for scientific research. They can be used to demonstrate the relevance of research results and their applicability in practice. Software tools enable to conduct case studies in scientific as well as in industrial environments. Furthermore they extend the addressees of research results and are often the root for industrial or open source software development projects.

Although the importance of software tools for research is well accepted in the scientific community the quality of scientific software is seldom adequate. Scientific software is often error prone, doesn't implement the intended functionality or is only usable by the developers since either the documentation is missing or outdated.

In our experience quality problems of scientific software are mostly caused by an inadequate software development process and an insufficient development infrastructure. Many scientific software projects don't define a dedicated development process or implement a process that doesn't cope with the specific constraints of scientific software development. However, this induces various problems in the management of the software and in the collaboration of the developers. Those problems often lead to severe quality issues in the software.

The remainder of this paper is structured as follows. In chapter 2 we characterize scientific software projects and delineate them from other development projects. In chapter

*Email addresses:* `vhoff@swc.rwth-aachen.de` (Veit Hoffmann), `lichter@swc.rwth-aachen.de` (Horst Lichter), `alexander.nyssen@itemis.de` (Alexander Nyßen)

3 we briefly describe ViPER, a scientific software development project that inspired most of the results presented in this paper. Chapter 4 defines a set of goals for an adequate scientific software development process for medium to large size, long living projects. In chapter 5 we introduce a process frame that enables those defined goals. Afterwards we introduce best practices for our process frame in chapter 6. Finally chapter 7 situates our best practice approach with some related research before we give some concluding remarks in chapter 8.

## 2. Characteristics of Scientific Software Projects

Scientific software projects are very diverse. They vary in size, lifespan, application domain and used base technology. Therefore it is impossible to define one single fixed development process that fits for all those projects. Anyhow, all scientific software projects can be distinguished by two major characteristics that clearly separate them from other development projects like open source or commercial development.

1. **Scientific software projects are embedded in one or more research projects.** This has various impacts to the software and its underlining process, because research directions may change based on obtained results, research opportunities or funding. This usually shifts the focus of the software and often induces changes of the requirements and architecture. Additionally not every research produces the expected results. Empirical studies may uncover misconceptions and flaws in earlier research. This often also impacts the software since empirical results may show that methods or technologies supported by the software are not feasible. Thus scientific software projects have to deal with dead-ends and rollbacks regularly.
2. **Scientific software projects are preformed with heavy student involvement.** Main parts of the functionality of many scientific software projects are developed in students' theses or with the help of student developers. This causes two special problems.
   First, students are usually inexperienced. Many students have never been in real world development projects. They often lack software engineering know-how and have no experience with tools, languages, libraries or frameworks that are used in the project.
   Second, students just take part in the development of the software for a very limited period of time. Students typically join the development team just for their thesis and are seldom a part of the development team for more than nine months. Additionally students often only contribute to a very limited part of the project, namely the specific feature that is being developed in the context of their thesis. Thus students often don't have overview over the project as a whole and have little interest in the overall project success.

Any development process aiming to support and guide scientific software projects has to consider these special characteristics. In the following however we concentrate on medium to large size, long lining development projects, like SESAM [20], Fujaba [7] or MontiCore [11]. I.e., we primariliy consider scientific development projects with a livespan of at least several years and at least 5-10 developers. Those projects typically create software, that should be used in multiple contexts, by several different kinds of stakeholders, especially those who are not directly involved in the development process. Although several of

the described best practices are also useful in smaller projects the development process described here is not adequate for small prototype projects that don't have to address documentation or architectural quality because it enforces different quality assurance measures that are unnecessary in those situations.

### 3. Viper - A Scientific Software Development Case Study

In the following we briefly describe the ViPER project that we run for several years. The essence and the lessons learned of this project are the central experience basis to propose a process frame for scientific software projects.

ViPER (Visual Tooling Platform for Model-Based Engineering) [21] is a tooling platform to leverage model-based engineering. It is based on Eclipse [6] technology and offers support for UML-based visual modeling, UML-based ANSI-C code generation, extended support for editing and simulating of detailed narrative use case descriptions, as well as built-in dedicated methodical support for the MeDUSA-Method [12].

Having started in July 2004 as a mere experiment to evaluate Eclipse related EMF and GEF technologies in the form of a simple UML state machine diagram editor, which was bundled into a single plug-in and accounted for about 3800 lines of source code, ViPER has grown into a rich and extensive tooling environment in the following years. It up to now bundles round 50 plug-ins and subsumes more than 200,000 lines of code, incorporating the contributions of 18 developers (scientific staff as well as students) over the years. In the recent years some parts of ViPER have been promoted to the Eclipse frameworks or to separate open source projects.

As ViPER grew the development infrastructure and the process management measures had to be adopted several times and nowadays ViPER is developed with a defined development process that relies on independent features, build on a common platform and a common development infrastructure.

The development infrastructure of ViPER consists of a CVS-Version-Management Repository [5] and a Bugzilla-Issue-Tracker [4], that are connected by a SCM-Bug [19] integration. Additionally ViPER maintains a dedicated build server for release engineering and quality assurance. This build server runs an ANT based Eclipse product build that creates releases on a nightly basis. Apart from the assembly of the ViPER-Product each build includes the validation of the source artifacts with checkstyle rules and the creation and execution of a regression test suite to assure functionality and conformance.

Inspired by Scrum [18] the ViPER project defines a lightweight sprint oriented planning process and uses Bugzilla as planning tool. Every sprint is connected to a target milestone in the Bugzilla system and all intended functionality is filed as enhancement requests against this milestone.

Since the ViPER development team is quite small the ViPER process relies on direct communication and the coordination of different features is done in weekly management meetings. Those management meetings are the core activity of the ViPER process and must therefore be attended by all project members. They are used for planning, the coordination of feature projects and the maintenance of the platform. They regularly last for about 1,5 hours. Every team meeting is started with a short status report from every project member. Afterwards team meetings include a management part, where sprints and scoping sessions are planned or development tasks are assigned. The meetings often end with an open discussion session where specific problems are discussed.

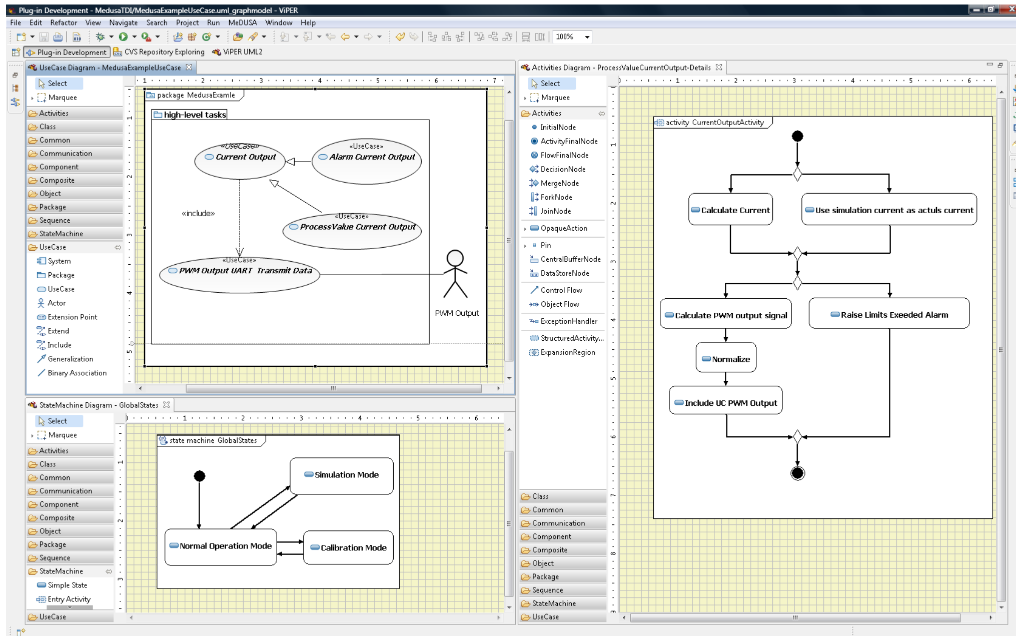



Figure 1: ViPER Screenshot

Currently ViPER is managed in a well established process that supports its efficient development. Although it is lightweight enough to enable an easy setup of new feature projects it includes measures to assure integration and quality. Moreover parts of the ViPER process have been successfully transferred to several other scientific development projects. In the next sections we present the essence of the ViPER process in the form of an extensible process frame and a set of best practices for scientific development projects.

## 4. Objectives of a Scientific Software Development Process

As stated before we primarily target long living scientific software projects that should be used by several different kinds of stakeholders. The primary objective of a development process for this kind of scientific software projects is to enable the development of software in sufficient quality. For most scientific software projects a beta-quality level is sufficient, since it is neither possible nor necessary to develop the software to full commercial quality. Anyhow, since scientific software must be usable in case studies and field research they have to provide a minimal level of usability and robustness. Apart from implementing the intended features correctly the software must be documented and robust against simple misuses like invalid input data.

Additionally the architecture is a core concern for many scientific software projects, since the software is often enhanced by new features or restructured due to new requirements. Thus the maintenance of sustainable software architecture is a vital task for a scientific software development process.

To achieve those objectives the process must especially address the specific characteristics of scientific software projects presented in chapter 2. This implies that a feasible development process must provide support for agile, evolutionary development, which is induced by the volatile development. Changes caused by shifted tool usage and the impact of requirement changes to the architecture must be manageable, too. Furthermore the software has to be rolled back to defined baselines after reaching a dead-end in the project.

Additionally a development process must address the specific human factors in scientific software projects, which means it must enable students to produce high quality results and to contribute to the project. Thus a development process must define clear rules and provide measures to transfer existing and new know-how. Producing high quality results however demands a lot of discipline especially from inexperienced developers like students. Therefore the development process has to provide team measures to raise the involvement of the participating students and to raise their interest in the project success.

## 5. A Process Frame for Scientific Software Projects

As denoted before scientific software projects have two specific characteristics that affect their lifecycle and development processes. First, they undergo continuous changes since they evolve alongside the theoretical research. Second, most of the functionality is developed in different thesis works performed by students.

In this section we describe an iterative, incremental and evolutionary process frame for scientific development projects that explicitly addresses those specific characteristics. This process frame should be enhanced with best practices described in the later sections of this paper.

Scientific software projects resemble open source projects in many ways. They are evolutionary, open, focused on an extensible infrastructure to react on usage shifts, suffer from quick changes in the development team and mainly develop functionality in dedicated subprojects. Therefore we have adopted many ideas from open source development projects for the proposed process frame and the best practices.

Our process frame reflects the overall project organization approach of incremental development projects based on feature development. A feature may be defined as a set of related functions that are implemented together to realize a specific goal. Furthermore it explicitly considers the systematic development of reusable features, which we call platform.

To maintain a stable feasible architecture and manage the development of the various features, the process frame consists of three separate but interconnected sub-processes: a platform-process, a set of feature-processes and an underlying coordination and collaboration process (see Figure 2). We will explain these processes in the following.

### *5.1. Platform Process*

The objective of the platform process (similar to project line development) is to ensure that a reusable platform (called the platform feature) is developed and maintained continuously. The platform process is initialized by the platform team at the start of a scientific development project and is performed until project end. It manages the

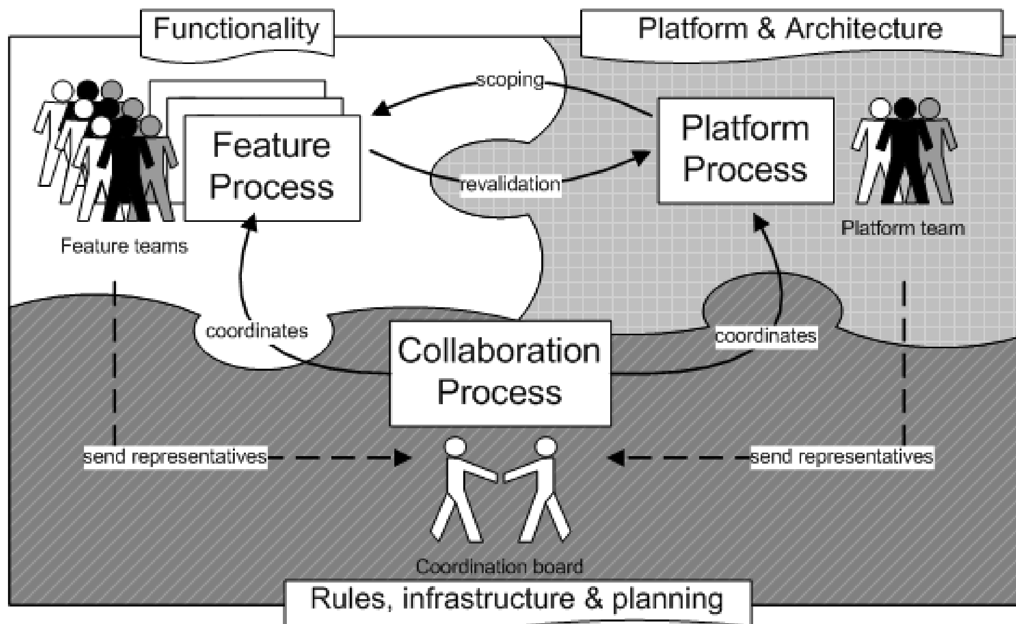



Figure 2: Dependencies between the sub-processes

platform feature, a single feature that contains two different kinds of information. First, it contains the software's common code platform (e.g. reusable functionality that is used as a basis for feature development). Second, it manages the software architecture. It defines an architectural pattern for the entire software and a common scheme for the definition of interfaces between features. Additionally it introduces a set of tasks for the maintenance and quality assurance of the software architecture, e.g. scoping, revalidation or measurement tasks. Because the platform-feature is the core of the development and the platform process is performed continuously, the platform team should consist of experienced developers that are responsible for the project and accompany the project for a longer time. Thus, in most cases the platform team is formed by PhD students, post-docs or assistant professors.

*5.2. Feature-Process*

As the main functionality of the software is developed in feature projects, a new feature process is initialized for each feature development. Because every feature process is performed by an autonomous feature team, several feature processes can be run in parallel. The single feature processes may vary broadly, since the feasibility of a development process for a feature is dependent on the type of the feature, the abilities of the team, the schedule and the workload. But it has to be ensured that each feature process includes an internal planning for the feature, a requirements engineering session, the definition of the feature's architecture as well as the feature's implementation, integration and documentation.

*5.3. Coordination and Collaboration-Process*

Finally, the coordination and collaboration process manages and coordinates the evolutionary feature based development of the software. It defines all process standards, i.e., process rules and documentation standards and templates for the platform and all feature processes. Furthermore, it defines and establishes the development infrastructure, constituting of e.g., configuration and change management, release engineering infrastructure and development environment. Moreover, the coordination and collaboration process determines the project's global time schedule, i.e., it defines synchronization points for the platform and all feature projects. Finally, it nails down global management tasks for the coordination of development teams. The coordination and collaboration process is typically managed by a coordination board that is created out of developers from the feature and platform teams.

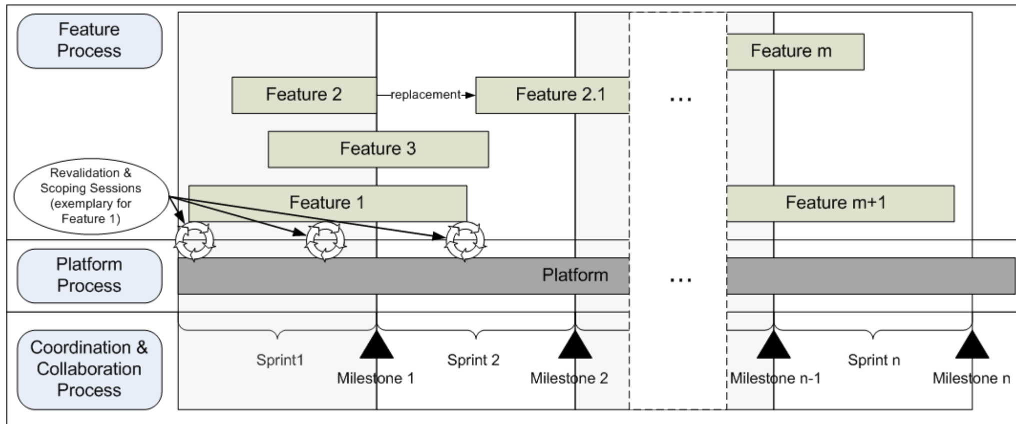



Figure 3: Project snapshot showing sub processes on the timeline

Although each feature process may define its own specific development lifecycle, all feature processes as well as the platform process must adhere to the coordination and collaboration process. They must apply the defined development infrastructure and the documentation must conform to the templates. Additionally the coordination and collaboration process defines the project pulse, i.e., it defines the releases and corresponding milestones that are obligatory for all feature processes.

The presented process frame on the one hand enables to efficiently develop features in dedicated feature processes, since feature processes are lightweight an rely on a common infrastructure. On the other hand a global, sustainable platform can be developed and maintained. Thus a set of management practices must be established to coordinate the decentralized sub processes of the presented process frame. A set of best practices addressing especially the most pressing management challenges is presented in the following section.

## 6. Best Practices

In the following we present a couple of best practices that we recommend for scientific software projects. These best practices are often adaptations of agile or open source

techniques and are mostly associated to the platform-process or to the coordination and collaboration process.

*6.1. Release and Version Management*

*Apply a sprint based Release Management*

Industrial strength release planning is typically inadequate for scientific software projects where features can quickly emerge and disappear again, because it is too heavyweight and inflexible. However, the delivery of the features and their integration in the platform needs to be managed. We therefore recommend a lightweight release planning similar to sprint planning in Scrum [18]. We recommend sprints of about three months. Although this is longer that typical Scrum sprints we have very positive experiences. First, the development speed is often slower compared to commercial projects because often developers (students) are inexperienced and the feature-teams are very small. Second, we recommend that each milestone should include contributions from every active feature-project. This positively raises the acceptance of milestones as the most important delivery target. But this can only be achieved if the sprints are not too short. At the end of each sprint a release is build which should contain all completed features as well as those features under development that have reached a stable state.

*Use an integrated Change Management and Version Control System*

Scientific projects are performed in a decentralized way and evolve evolutionary. Therefore the management of changes and their impact to existing project artifacts is a crucial task. This demands a close integration of the version and change management system. Every bug or enhancement request should be handled by a ticket of the change management system and every change to a project artifact must be associated to a respective ticket. This enables to keep track of changes and to trace changes back to project artifacts. Moreover all artifacts (including source code, documentation, test cases and examples) should be under version control. This has three mayor benefits. First, storing all kinds of documents in a version control system reduces loss of important information since even less important information is stored. Second, the actuality of non-code artifacts, like requirements or architecture definitions, is raised significantly. Third, monitoring the impact of changes to all affected artifacts is supported. Additionally we recommend to version all artifacts of a single feature project together.

*Implement all new features against the head revision*

Each new feature should be developed against the head revision of adjacent feature projects and the platform project. This demands some discipline from the developers, since changes to the interface of one feature may directly affect others. Thus a stable interface design and continuous communication are needed. We experienced only little communication overhead if the interfaces are defined upfront and described explicitly. Besides, the integration effort of components developed in parallel is reduced. Additionally an immediate integration of features under development has positive effects on the team spirit, since developers get a quick feedback about their results.

*Use an automated build and test system*

An automated build system should be used to assure the integration of all features including those that are under development. We recommend a dedicated build process performing nightly integration builds to assure the compilability and integration of the components. Additionally the build system should create release builds on a regular basis. We recommend one release alongside each sprint. A release should always be preceded by several release candidates used for manual testing.

Furthermore we recommend enriching the build system with automated regression tests and static analyses performed after each successful build. This is especially important since many developers only work on fragments of the system functionality and don't test the integration with other fragments or features. Although the integration of a dedicated quality assurance infrastructure in the build system needs quite a lot of setup effort, it is worthwhile since deficiencies and problems in the source code are detected very early.

*Provide an update infrastructure*

Scientific software often changes and evolves quickly. Therefore an update infrastructure should be provided once the software is released, e.g., for field studies. An operative update infrastructure improves the user's acceptance to use the respective software, since they can experience the evolution of the software and keep up to date with only little effort. Therefore a single available source for updates and software releases should be established and users should be informed about new updates.

*6.2. Quality assurance*

*Create regression tests*

The platform project and its feature projects change rapidly because requirements are changed or new requirements emerge. Moreover functionality is moved from feature projects to the platform or vice versa as a result of scoping sessions or performed refactorings. This demands to run regression tests as often as possible to assure the stability of the software. We recommend implementing black box regression tests for the platform and every feature. Moreover the coverage of those tests should be measured and additional tests should be added if the coverage is insufficient.

*Perform a quality assurance phase for each feature project*

Feature projects should plan a feature freeze and a quality assurance phase at the end of their development (often called endgame). The endgame should assure three important quality aspects. First, it must assure that the required functionality is implemented and that the feature is correctly integrated in the platform. Second, it should assure that the feature's architecture conforms to the platform's architectural rules. Third, the endgame should assure that sufficient documentation is available. Since feature projects are typically performed in the context of students' thesis works and students are often unavailable after the end of the thesis a rigorous endgame is vital for the overall project success. Thus any information that is not handed over is lost and can only be recovered with very high effort. Additionally we recommend that the platform team performs a platform scoping session as part of the endgame together with the feature team (see best practices of platform management).

*Measure the architectural quality on a regular basis*

The architecture is a core concern for most scientific software projects. To maintain a sustainable architecture in the dynamic environment of scientific software projects the conformance of feature implementations to the architecture definition in the platform must be measured and checked regularly. Nowadays, there are several tools to measure and assess software architectures [9] [10] that can be used. The results of those measurements should be discussed in team sessions to sensitize the developers for the architectural demands of the platform and thus to get less violations in upcoming sprints or feature projects.

*Explicitly define the interfaces of each feature*

The interfaces of every feature should be explicitly defined and access to any feature should only be allowed through those interfaces. Thus the impact of changes of a feature to other features is reduced and manageable. The adherence to this rule should be checked automatically alongside the quality measures performed by the build system.

*Define templates and rules for project artifacts*

A template and a respective set of rules for all kinds of project artifacts (code and non-code) should be defined. Templates increase the readability of documents and ease identification of changes if a version management system is used. We recommend to use style rules defining formatting and encoding for all code artifacts and to check those rules by an automatic checker.

### *6.3. Platform Management*

*Perform platform scoping sessions on a regular basis*

The platform contains reusable code that should be used by all feature projects. Its main purpose is to prevent feature projects from developing the same functionality several times. However, most of the functionality is developed in feature projects including reusable functionality that should be part of the platform. Therefore special platform scoping sessions should be performed regularly to transfer reusable parts from feature projects to the platform. A scoping session is a joint session of a feature team and the platform team and has two purposes. First, potential platform candidates useful for other features should be identified. Second, each candidate should undergo a detailed scoping analysis. This analysis has to define a migration strategy for the candidate, i.e., a set of refactoring and generalization steps that are necessary to integrate the candidate with the platform. Every migration strategy should afterwards be planned as a normal task in a sprint.

*Perform refactoring analysis before developing a new feature*

Whenever a new feature should be developed the first step is to evaluate the current platform architecture to ensure that the platform architecture is still adequate to implement the new feature. If this is not the case, a set of refactoring steps have to be identified to improve the architecture and to enable the integration of the new feature. These refactorings are performed (typically by the platform team) before the new feature project is started. This results in a stable and maintainable architecture and minimizes the integration effort of new features.

*Check for obsolete features on a regular basis*

Since resources in scientific software projects are short no effort should be wasted to develop or maintain features that are no longer needed. Thus it is necessary to check for features that are no longer needed regularly. A feature may become obsolete for three reasons. Either because the usage of the tool has changed or it was replaced by a new feature or it can be replaced by functionality imported from e.g., open source projects. We recommend to check for obsolete features at least every time a sprint is planned. For each obsolete feature the platform team has to define a decommission strategy. It defines necessary steps how to migrate those features depending on an obsolete feature to the replacing one. The implementation of the decommission strategy should be planned as a normal task in the upcoming sprint, similar to a migration strategy.

### *6.4. Team Management*

*Perform pair programming sessions with new developers*

Inexperienced developers should at first do some pair programming session together with an experienced developer (often a member of the platform team). This is highly accepted and has three mayor benefits. First, it enables the new developer to bridge the technological gap. Second, the experienced developer can transfer knowledge about rules and standards to the new team member easily. Third, the integration of new team members to the development team is facilitated, since they get to know other team members better.

*Provide an infrastructure for knowledge documentation*

Every project should provide an infrastructure to document and transfer knowledge, e.g., an open wiki or a blog. This is extremely useful if the project is using a lot of open source technologies and frameworks where the documentation is scattered in the Web. Those systems serve as a starting point for developers to search for documentation, tips and advices.

*Perform regular team sessions*

The project team should meet on a regular basis. Those meetings are a central practice in our experience and have four mayor purposes. First, they are a management and planning instrument. In the team meetings every developer should present the status of the current work and the next upcoming development steps, to keep track of the overall project status. Additionally new sprints should be planned in the meetings. This raises the developers' commitment to deliver in time, because they are involved in the planning. Second, team meetings are a discussion platform for specific problems of the developers which has the following benefits. First, whenever a developer is stuck with a problem the team may help him to find a feasible solution and second the discussions can be used to spread technical as well as process knowledge. The third purpose of team meetings is to maintain the platform architecture. Therefore the results of architectural measurements should be discussed, architectural decisions should be made and scoping sessions should be planed. The last and maybe the most important purpose of the team meetings is to create a team spirit and raise the developers' involvement in the project. Therefore team meetings should have a casual ambience and encourage open discussions.

*6.5. Best Practices in Context*

Obviously some of the presented best practices have different relations to each other. Some are prerequisites for others or the implementation of one practice positively influences others. Figure 4 shows a sketch of those dependencies.

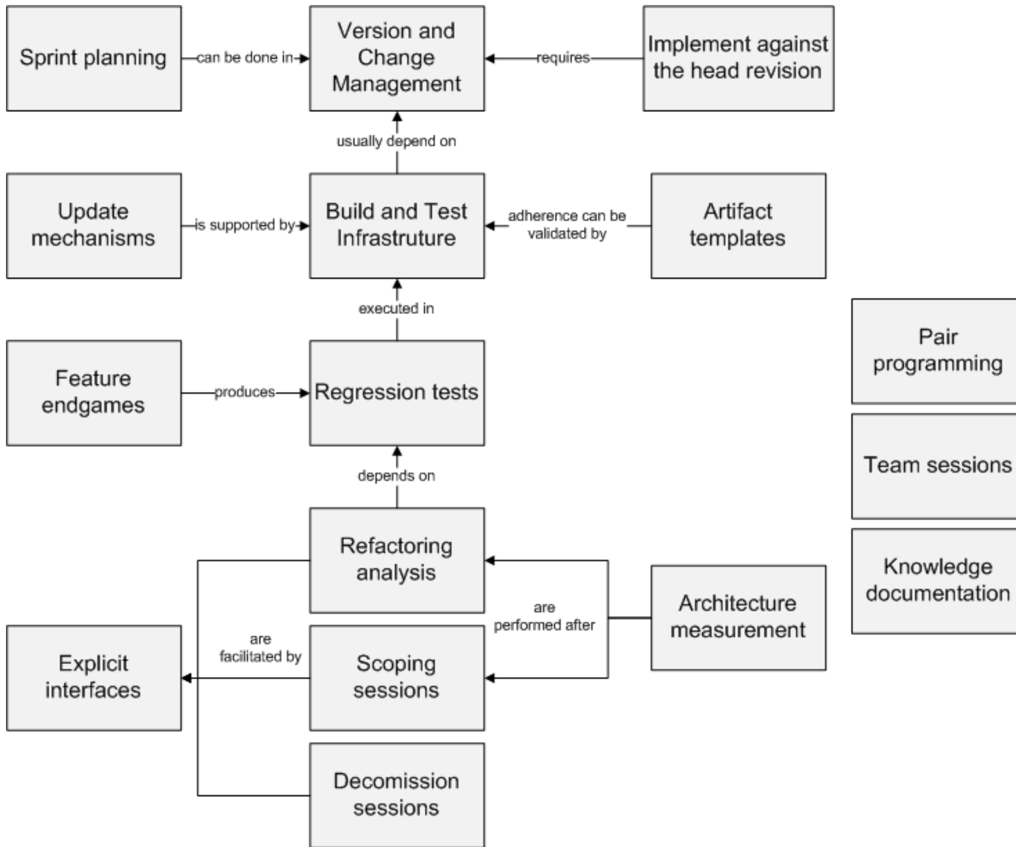



Figure 4: Best practices in context

These dependencies obviously influence the sequence to establish practices in a project, typically practices should be introduced in related groups.

## 7. Related Work

Feature oriented development processes have been discussed in several publications. According to the feature driven development approach proposed by [13] all features are defined upfront. Afterwards all features are developed in an iterative fashion, where in each iteration one feature is realized. Several agile development processes like Scrum [18] or XP [2] use features for the planning of iterations but during devlopment all features of the iteration are developed together by one development team.

The product-line engineering community [14] introduced the idea of a common platform and scoping techniques for its maintenance. However, those approaches mainly focus the management of variability of multiple products. Thus they are not specific enough for scientific software development projects with only one single product, which consists of a platform and several features.

Best practices for development processes have be discussed by several authors. [3] e.g., focusses mainly human aspects in development projects whereas [1] describes a quite elaborated set of process patterns for large-scale object-oriented systems. In resent years several open-source projects like Gnome [8] or Eclipse [6] have evolved to software ecosystems where one project builds the platform for the development of several others. Most of those define a development process with a milestone oriented release planning and best practices. [16] discusses different management aspects of those projects. However, those projects don't aim for one integrated solution. Thus no exlipicit scoping and revalidation of the platform is planned.

Today none of the current development approaches explicitely concerns the specific characteristics of scientific development projects. Although many of the discussed approaches and techniques can be usable in scientific development projects most of them need to be adopted to those projects specific needs.

## 8. Conclusion and Outlook

In this paper we discussed the importance of a feasible development process as basis for the creation of high quality software in long living scientific development projects and we depicted the specific aspects of scientific software projects that impact a feasible development process. We outlined a generic process frame, that addresses the specifics of scientific software projects and sketched a set of best practices for the presented process frame. Furthermore we presented the ViPER project as one case study for a successful scientific development project, that was managed according to the presented best pratices.

We have sucessfully performed the QMetrics project [17, 15] with the described best practices and we are currently running several other scientific projects we perform at the faculty and with different industrial cooperation partners, with the presented process frame and the described best practices. Our experience in all these projects are also very positive. Additionally we plan to do a systematic analysis of the process and its respective best practices. Therefore we intend to perform a series of questionnaire oriented qualitative analyses in several scientific projects that may or may not implement our process frame.

Until now the described process frame only considers generic basic best practices. In the future we plan to enrich the best practice section with a set of optional measures for specific project settings. Moreover we plan to provide strategies for the introduction of a best practice based process frame in a new project or the migration of a running project to such a process.

## References

[1] Ambler, S. W., 1998. Process patterns: building large-scale systems using object technology. Cambridge University Press, New York, NY, USA.

[2] Beck, K., Andres, C., 2004. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional.
[3] Brooks, F. P., August 1995. The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition), 2nd Edition. Addison-Wesley Professional.
[4] Bugzilla Project, Jun. 2010. Bugzilla - project website.
URL http://www.bugzilla.org/
[5] CVS Project, Jun. 2010. Concurrent version system - project website.
URL http://www.cvshome.org/
[6] Eclipse Project, Jun. 2010. Eclipse - project website.
URL http://www.eclipse.org/
[7] Fujaba Project, Jun. 2010. Fujaba - project website.
URL http://www.fujaba.de/
[8] Gnome Project, Jun. 2010. Gnome - project website.
URL http://www.gnome.org/
[9] Hello2Morrow, Jun. 2010. Sotoarc - project website.
URL http://www.hello2morrow.com/products/sotoarc
[10] Metrics Project, Jun. 2010. Metrics - project website.
URL http://metrics.sourceforge.net/
[11] MontiCore Project, Jun. 2010. Monticore - project website.
URL http://www.monticore.de/
[12] Nyßen, A., Lichter, H., Streitferdt, D., Nenninger, P., 2008. Medusa - a model-based construction method for embedded and real-time software. In: COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference. IEEE Computer Society, Washington, DC, USA, pp. 1376–1382.
[13] Palmer, S. R., Felsing, J. M., February 2002. A Practical Guide to Feature-Driven Development (The Coad Series). Prentice Hall PTR.
[14] Pohl, K., Böckle, G., Linden, F. J. v. d., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
[15] QMetric Project, Jun. 2010. Qmetric - project website.
URL http://www.qmetric.org/
[16] Sandred, J., 2001. Managing Open Source Projects: A Wiley Tech Brief. John Wiley & Sons, New York, NY, USA.
[17] Schackmann, H., Jansen, M., Lischkowitz, C., Lichter, H., 2009. Qmetric - a metric tool suite for the evaluation of software process data. In: ICSE Companion. IEEE, pp. 415–416.
[18] Schwaber, K., Beedle, M., February 2002. Agile Software Development with SCRUM, illustrated edition Edition. Prentice Hall.
[19] Scmbug Project, Jun. 2010. Scmbug - project website.
URL http://freshmeat.net/projects/scmbug/
[20] SESAM Project, Jun. 2010. Sesam - project website.
URL http://www.iste.uni-stuttgart.de/se/research/sesam/
[21] ViPER Project, Jun. 2010. Viper - project website.
URL http://www.viper.sc/

# From Design to Tools: Process Modeling and Enactment with PDE and PET

M. Kuhrmann[a], G. Kalus[a], M. Then[a], E. Wachtel[a]

[a] *Technische Universität München, Institut für Informatik – Software & Systems Engineering, Boltzmannstr. 3, 85748 Garching, Germany*

**Abstract**

In our research group we work on software development process modeling and particularly deal with the question how a defined development process can be brought to life in a project. Modeling of complex processes with hundreds of elements requires comprehensive tool support. Successful enactment of a process during project lifetime does depend on how well the process is supported by tools for project execution – among equally important factors that deal with psychological and organizational questions. To prove ideas developed in research and to ease our day-to-day work in industrial cooperations, we are developing many tools ourselves. In this paper, we present the *Process Development Environment* and the *Process Enactment Tool Framework* that support intuitive modeling of development processes and their enactment. We give an overview over the organization and the process of our tool development, challenges we are facing, and present the lessons learned from academic tool development both from the researchers' and the students' point of view.

*Keywords:* meta modeling, process models, enactment, development organization, student team, academic tools

## 1. Introduction

"Nice work with the process, but how does it support me during the project?" – it is such questions that we are facing in our day-to-day work with partners from industrial practice. Most of our projects deal with software development process design and enactment, especially with the question how a defined process can be applied and brought to life in a project.

The software development processes that we deal with are based on formal meta-models. Not only are such processes perfectly suited to be supported by tools (e.g., for derivation of project plans, generation of artifact templates and documentation), tool support is a necessity during design (the German V-Modell XT process XML description file for example is 4.8 MB big with a couple of ten thousand elements). Some tools are available that partly suit common needs. Yet, many needs coming from industrial projects or ideas we would like to try out in the context of our own research are not addressed. We are therefore developing many special-purpose tools

*Email addresses:* kuhrmann@in.tum.de (M. Kuhrmann), kalus@in.tum.de (G. Kalus), then@in.tum.de (M. Then), wachtel@in.tum.de (E. Wachtel)

ourselves. These range from "quick and dirty" micro-tools up to comprehensive frameworks with a quality level close to commercial solutions.

The tools that we are building cover all stages of the life cycle of a development process. We have tools for definition, design, and validation of process models. We have also built process-aware tools for project execution. Our long-term goal is to cover the whole life cycle of the software development process from process design to its enactment with seamlessly integrated methods and tools.

*Challenges.* Because our software is motivated by industrial cooperation projects and in use by industrial partners, we have to maintain certain quality standards with our software. However, as a university, we are no solution provider and do not have a professional development staff. In fact, we do not have "real" development resources. Most of the development work is done with and by students. This fact greatly influences the way that we (must) organize tool development. Some noteworthy implications are:

- It is not easy to find students.
- The duration of student theses is limited.
- A thesis must contain conceptual and theoretical work.
- It is hard to assess the qualification and suitability of students.
- The results vary in quality and in how easy they can be integrated into our tools.
- Researchers and students sometimes have different goals.
- It is difficult to establish a meaningful development roadmap.
- Our selected technology is usually not taught in lectures.

*Outline.* Before explaining in greater detail the specific challenges with tool development in our research group in Section 3, we present the tools themselves in Section 2. We are covering only the tools developed at Technische Universität München in the research group *Development Processes*[1]. Other research groups at our chair are also developing tools of considerable size and are facing similar challenges. In Section 4, we give an overview over our group's organization that is a direct consequence of our experience with academic tool development during the last years. In Section 5, we describe the transfer strategy to promote our software. The paper is closed in Section 6 with a discussion and open questions with regards to academic tool development.

## 2. Process-aware Tools

Historically, requirements coming from industrial cooperation projects as well as questions from our own research work have lead to the development of a couple of prototypical tools that were only used by ourselves. Over the years and many student projects later, some of

[1] The chair for Software & Systems Engineering is organized in several competence centers. Each has a focus to a particular area of Software Engineering. The emphasis of the competence center *Development Processes* is the modeling of variable development processes and their systematic enactment.

the tools have outgrown the experimental stage and are now covering large parts of the life cycle of a software development process. In this section we first give a short impression of how we understand development process models and what we consider important for modeling and enactment of such processes. We then present our two flagship tools *PDE* (Section 2.2) and *PET* (Section 2.3). Finally, we give a short overview over some other (smaller) tools that we have developed or are developing.

### *2.1. The Life Cycle of a Development Process and its Modeling*

There is a large variety of software development processes and methods, reaching from small "agile" methods to comprehensive structured process models. The process models and methods themselves are not in the scope of this paper and we refer to previous works [8, 2, 14] for an in-depth discussion. Whatever the flavor of the process or method, it usually cannot be used out of the box. It must be customized to the special needs of an organization or project and adequately supported by tools for project execution. To facilitate customization, most process models have concepts and rules for tailoring. Inspired by software product lines [12], so-called *process lines* [21] describe a set of stable core process assets (commonalities) that are mandatory to all projects plus optional process assets (variabilities) to be applied in specific project contexts. For the enactment of processes in tools [16], process models based on formal meta-models are of greatest interest to us.

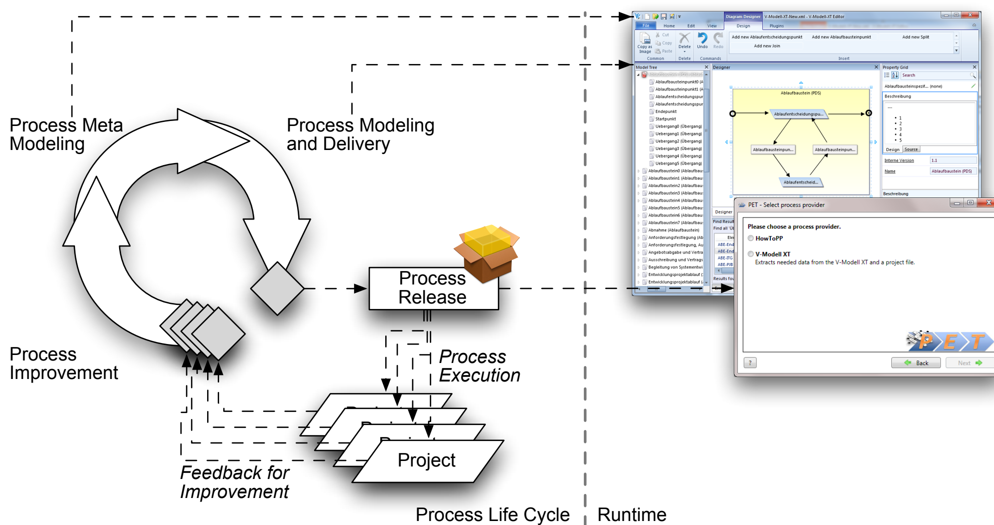



Figure 1: Process life cycle with associated tools

The life cycle of our process models is described by Figure 1, based on [15]: (1) Process meta-modeling defines fundamental process model language structures using techniques such as XML-Schema or custom domain-specific languages [9]. (2) In the second stage the process is modeled (creating structures e.g. work product milestone associations and content e.g. activity descriptions). This stage is finished with a release of a process, which is delivered for (3) application in projects. A feedback loop is established, which opens the door to (4) continuous

3

process improvement [13]. The tools we are developing are closely aligned to the development process's life cycle.

*Implementation Technology.* We have made the strategic decision to implement all our tools on the same platform. We chose Microsoft .NET with C# for *all* our tools. One single implementation technology for all our tools makes integration (relatively) easy. The ability for integration of our software is important because the students developing the software come and go and we have to make sure that their work does not depend on others to complete. Source code as part of a thesis is therefore normally developed in isolation and integrated after completion to create an integrated tool chain that comprises all components for the different stages of the development process life cycle.

### *2.2. Process Modeling: the Process Development Environment*

The *Process Development Environment* (PDE) [25] is a tool for the design of process models. It is based on *domain-specific languages* (DSL) [9, 3] to (1) describe the concepts of the process that the tool shall support and (2) to generate the development environment for process engineers. PDE provides functionality beyond what standard tools are capable of. In particular, it supports process model families or process lines [21] and offers a couple of custom views that should help the process engineer to deal with the complexity of process models.

*Context.* The design of a comprehensive, flexible, and customizable software development process is a complex task, which requires adequate tool support. Tool-supported process design should include not only modeling capabilities for the process model itself but also the ability to find inconsistencies and to generate feedback to the process engineers similar to the instant feedback provided to programmers by modern IDEs. Inconsistencies in a process model can be inaccurate or missing relationships, which, based on the complexity of the model (model element and dependency structures), can be difficult to find and, if not found, can have fatal consequences at later project stages. The development of PDE was motivated by practical experience gathered in projects with industrial and public service partners where we modified and customized the V-Modell XT standard[2]. The standard tools coming along with the V-Modell XT make modification and customization work cumbersome as those tools provide no visual representation of the higher-level model structures one would normally want to work with, nor is there any support for finding inconsistencies. Errors are communicated in a cryptic manner, if at all, with the error messages typically saying not much more than "Element is Null". As the XML-model of the V-Modell XT contains a huge number of elements and relations, locating errors and finding their source is hard and time-consuming. PDE was therefore developed for our own convenience with a strong focus on the V-Modell XT. The reliance on a DSL allows us to implement constraints and validation rules into the tool. In addition to the original V-Modell XT version we have in the meantime developed a variant for the proprietary development process of T-Systems. We are using this variant of PDE in a currently ongoing cooperation with T-Systems.

[2]The V-Modell XT [6, 7] is the German standard development process model for IT-projects in the public service area. The meta-model [23] is implemented using a XML-schema.

*Overview.* PDE [25, 26] uses the Microsoft DSL tools [3]. The graphical modeling application for the process engineer is generated from the DSL description of the process model. In general, the application is inspired by well known integrated development environments (IDE) for programming but with respect to process engineers' special needs. PDE provides a number of key features, such as:

- Language-specific validation including design time feedback.
- Multiple, language-specific graphical views (language-integrated or plugins).
- Plugin interface for additional functionality or graphical views.

*Architecture.* PDE's architecture [25] basically consists of a family of (meta-)models that are implemented by DSLs as shown in Figure 2. The core model of PDE is the *Language Meta-Model* that describes the domain of software development processes, used to specify the *Domain Meta-Model*, which is the representation of a concrete (process meta-) model (e.g. SPEM). The domain meta-model's structure is further used to automatically generate the *Domain Model* as an instance of the domain meta-model for integration with the *Tool Framework*. The tool framework is an extensible framework used to visualize and modify the domain model. A DSL created with PDE can host multiple view such as model trees, property grids, graphical designer surfaces or error lists (see Figure 4).

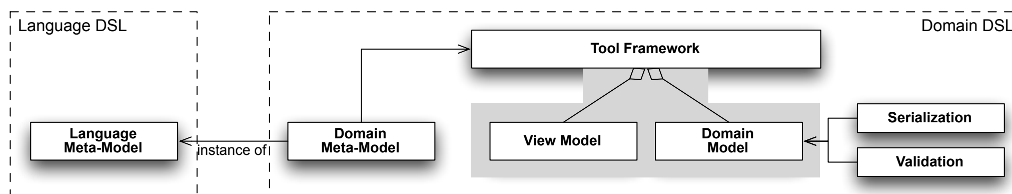



Figure 2: DSL Model Family for Process Modeling

Because of the multi-layered architecture with each DSL serving a specific purpose, the implementation of a new process model requires a couple of steps shown in Figure 3:

1. Specification of the domain meta-model.
2. Generation of the domain model and integration into the tool framework.
3. Applying optional extensions.
4. Building the editor.

The first step is done using the *Language DSL Designer*, which is a Microsoft Visual Studio extension. If the meta-model of the process is already explicitly written down, this step means to translate the meta-model of the process to the domain meta-model of PDE. In the case of the V-Modell XT, this first step involved the translation of the V-Modell XT's XML schema definition to the domain meta-model. The second step generates code from the domain meta-model using T4 templates [22]. The generated code can either be used as is or it can be extended – for example with custom validation rules or extensions (step 3). The "language engineer" can decide on the degree of modeling and which extensions will be available to the user. It is possible for instance to implement validation using functional programming languages such as F# instead of modeling

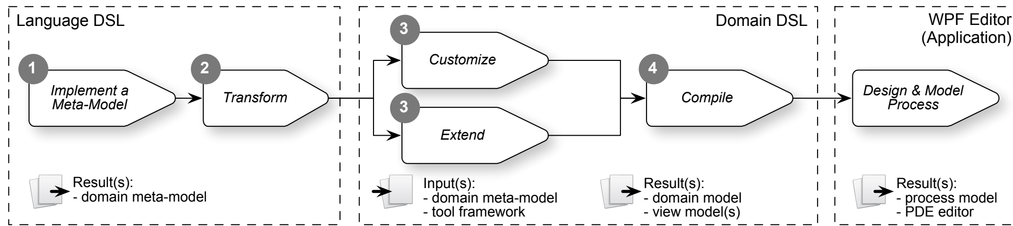



Figure 3: Concept of developing DSLs with PDE

a constraint (due to the ability of the .NET platform to mix several languages). An example of a coded validation rule is a checking algorithm for a graph being free of cycles. A plugin for PDE can be basic and only define additional functionality or it can provide custom views and editors that are integrated in the PDE main application. After customization and extension, PDE is ready to build in step 4. The result is the PDE process editor application (in Figure 4) for the specified model.

*Research Questions.* PDE is a fully functional application with functionality already going beyond the possibilities of standard tools. Yet, there is plenty of room for further work: Currently, validation constraints are specific to the process model. It would simplify the development of variants for new process models, if validation rules defined in a generic validation language could be reused across models. The same observation is true for serialization: currently, serialization rules have to be specified for each element individually. Another aspect is that it may be useful to combine several models into a larger model. A process engineer may then develop process components and assemble those to the overall process (similar to method libraries in the Eclipse Process Framework [4]). We have yet to gather practical experience on how to deal with model evolution and how it impacts the DSL-based generated working environment.

*Implementation Statistics.* As of June 2010 PDE had a code size of 40 KLOC. Additional 320 KLOC are generated by the T4 templates from the DSLs, adding up to a total code size of 360 KLOC. The readability of the generated portions largely depends on how well the domain model is modeled. If meaningful element names are chosen for the model, the associated source code is human-readable. However, generated and custom code are strictly separated through .NET's mechanism of partial classes, so a developer usually does not have to bother with the generated code other than for debugging of T4 templates.

### *2.3. Process Enactment: The Process Enactment Tool Framework*

The *Process Enactment Tool Framework* (PET, [17]) is a process transformation framework that bridges the gap between the defined process and project execution. It provides a generic process intermediate model, which arbitrary processes can be mapped onto and which transformed data can be extracted from. PET applies ideas and concepts of model-driven design & development to process enactment. The input to PET is a machine-readable process description (data model); its output side is usually a *process-aware* tool that provides some kind of process support to users during project execution. PET thus makes a contribution to the enactment of processes.

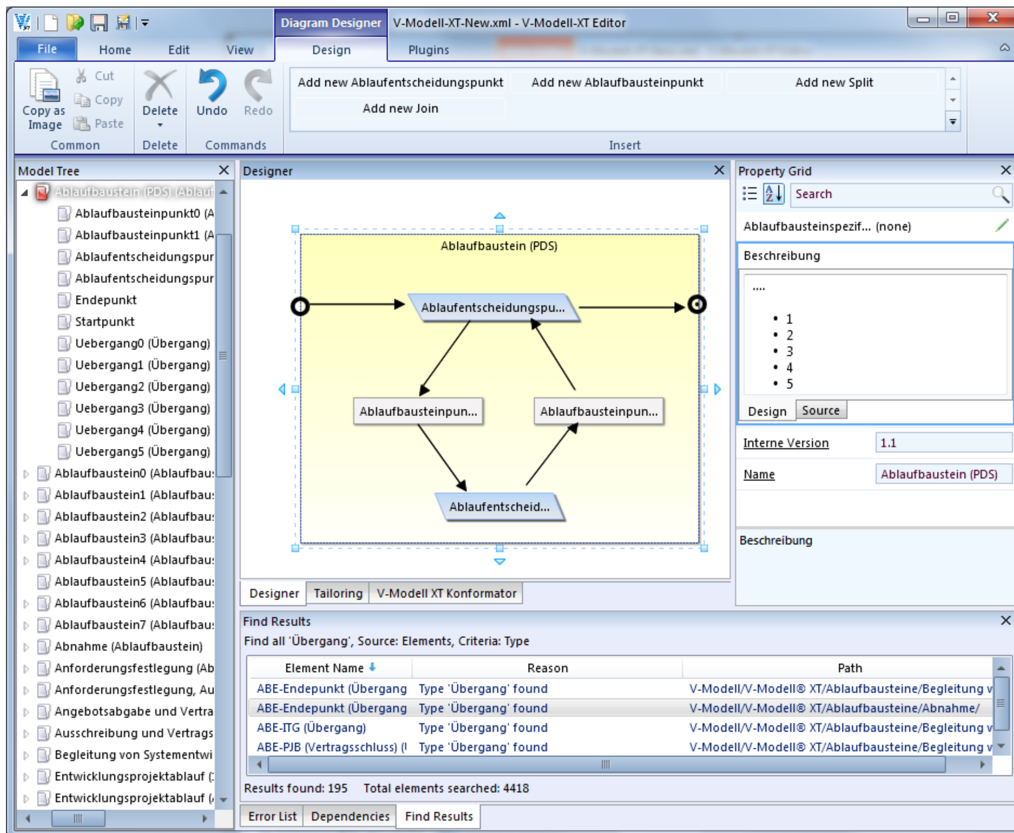



Figure 4: V-Modell XT DSL

*Context.* Tools can be helpful with regards to acceptance and enforcement of a software development process by providing assistance and automation [16]. Certain tasks prescribed by a process can be automated – relieving the team of repetitive process-related work. Many tools provide rich automation and assistance capabilities that can be used to make the process transparent to users (where adequate). Examples for such tools are Microsoft Team Foundation Server [18], Microsoft SharePoint [27] and IBM Rational Jazz[3]. The challenge is to enable these tools to access development process data and interdependencies to support users in their work relieving them of "process-only" tasks. A process defines a common terminology and thus is a medium for communication in a project. With regard to tool chains (integrated and loosely coupled ones) it is therefore essential that *all* tools "talk" *the same* process. Because different tools focus on different aspects of the overall development process, a challenge is to extract from the process model exactly the information relevant for a particular tool in a consistent manner, yet ensuring that data exchange between tools/tasks is possible.

[3]http://jazz.net/

*Overview.* The feature distinguishing PET from other transformation frameworks is that it decouples the tasks *process parsing* and *output data generation* by design. Plugins that read process models are called *process providers*, whereas tool data-generating plugins are named *tool providers*. Through this separation, it is possible to use the same process model parsing logic to create input data for different output data generators. Consequently, one output generation logic works for any process. Between the input process model and the output generator, PET introduces a generic meta-model that process models are mapped onto. This so-called *intermediate/core meta-model* contains common process elements, such as roles, work products, activities, milestones including their semantics and their interrelations. The intermediate meta-model was not designed to accommodate every possible element of process models but as a lean common subset. Thus, not every construct of a process may be mappable onto the intermediate meta-model. However, the elements it does define are sufficient for process enactment using tools (the main use case for PET). At the time of this writing PET process providers were implemented for:

- V-Modell XT 1.3.
- SPEM-based processes (currently under development).

Tool providers exist to generate:

- Process templates for Microsoft's Team Foundation Server 2008/2010.
- Project portals including document templates for SharePoint Portal Server.
- Word 2007/2010 document templates in the Office Open XML (.docx) format.

*Architecture.* The architecture of PET [17] is shown in Figure 5. The framework's architecture contains PET's *core*, *process providers*, *tool providers*, and an *application frame*.

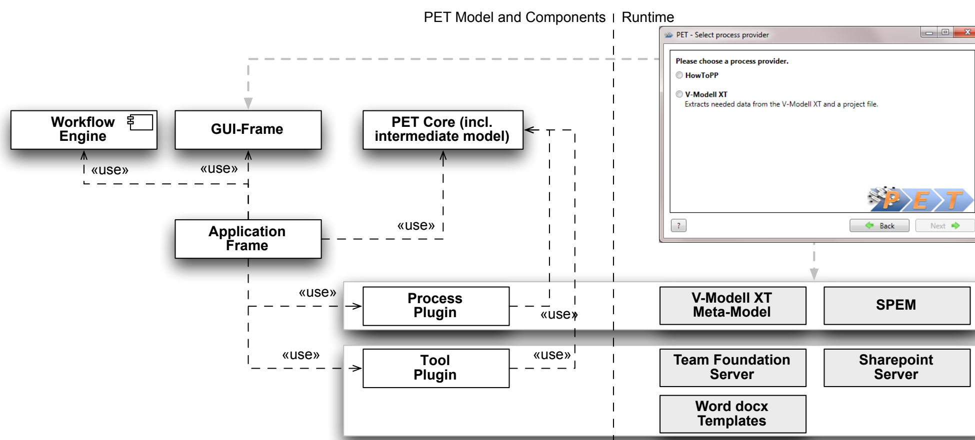



Figure 5: Schema of PET's architecture.

The PET *core* is at the center of the framework. It contains the interfaces and methods that are used by all process and tool providers as well as by the application frame. The core includes

the intermediate/core model, the process- and tool provider interfaces, and the conversion process interface, which gives plugins some control over the transformation steps. Process providers bridge the gap between the input process and the intermediate model. They are responsible for reading a process model and mapping it onto the generic PET process model, while preserving the process' semantics. Tool providers are responsible for the generation of the output data of the process transformation. Their input is the intermediate model, which was initialized before. Because parsing and (code) generation is encapsulated in self-contained provider plugins, a plugin developer can use arbitrary existing frameworks for parsing and output generation. Another central component is the application frame. It orchestrates the execution of the different transformation steps displayed in Figure 6. The graphical user interface allowing the selection of the different provider plugins and a couple of infrastructure components such as plugin configuration serialization and deserialization are also part of the application frame.

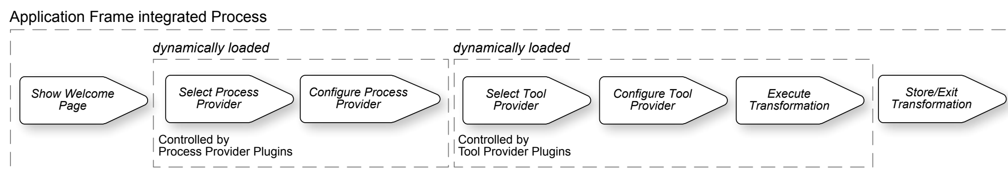



Figure 6: PET transformation workflow.

*Research Questions.* As already mentioned, there are two process providers available for PET (V-Modell XT [6] and SPEM [19]) at the moment. It would be interesting if other process models can be processed in equal manner. As mentioned above, the intermediate meta-model contains common elements of process models. Special elements of a process model possibly would get lost in the mapping process. We have to gather further experience with regard to extensibility of the intermediate model. Already, it is possible for process providers to add "custom" elements and non-standard semantics to the intermediate model. The downside of this approach is that tool providers will not understand the semantics of these elements by default and probably ignore them (a problem well known in the UML/MOF world [20, 1, 10, 11]). Allowing process providers to extend the intermediate model limits the set of tool providers that can be used with a certain process provider. The associated research question is how this can be done without contradicting PET's general transformation approach. PET's current architecture assumes the existence of exactly one process- and one tool provider per transformation. An idea we have yet to realize is the chaining of multiple tool providers, using the previously created output as input data. A scenario may be for example to first generate work product templates and then upload these to a SharePoint portal that will be used as project workspace.

*Implementation Statistics.* The PET code base has a size of 2.5 KLOC. The current version is around 15 months old. It is the result of an effort to combine a couple of independent transformation/generation tools that we had developed since 2005.

### *2.4. Further Tools*

The previous sections presented the main development lines – tools motivated by practice and in a state and quality near "real" products. Besides those, we developed small tools for (1) solving individual problems of our projects and (2) experimental tools.

In the first category we have add-ons supporting specific tasks during process design and implementation. Examples include assessing constructive process model conformance. Currently, it is available as early prototype, which will be integrated in PDE to support process engineers. We also have available specialized build tools supporting delivery of a customized process.

In the second category, we are currently working on an experimental work item tracking system that supports process design and execution, flexible work item structures, and arbitrary associations between elements [24]. Additionally, starting from this project, we investigate project runtime support. We are developing and testing assistance systems in the experimental work item environment to prepare for integration into PET. We do for example have an early version of a plug-in for Microsoft Word that provides additional functionality for PET-generated document templates.

## 3. Challenges in Academic Tool Development & Lessons Learned

In Section 1, we have briefly highlighted the special challenges that we are facing in academic tool development. To handle these challenges (discussed in detail in Section 3.1), we have found over time and through (sometimes negative) experience a couple of best practices, discussed in detail in Section 3.2. To provide an overview, Table 1 roughly puts in relation the challenges that we face and countermeasures we are applying.

| | (M1) **Advertisement** | (M2) **Payment** | (M3) **Publications** | (M4) **Project responsibility** | (M5) **Central repository** | (M6) **One platform** | (M7) **Promotion** | (M8) **Strategy** | (M9) **Lectures** |
|---|---|---|---|---|---|---|---|---|---|
| **Finding students** (C1) | X | X | | | | | | | |
| **Student fluctuation** (C2) | | X | X | X | | | | | |
| **Kind of student work** (C3) | | X | | | | | | | |
| **Qualification of students** (C4) | | | | | | X | | | |
| **Result quality** (C5) | | | | X | X | X | | | |
| **Goal congruency** (C6) | | X | X | X | | | | X | |
| **Development roadmap** (C7) | | | | X | | | X | X | |
| **Technology** (C8) | | | | | | | | X | X |

Table 1: Challenges and strategies

### *3.1. Challenges*

The greatest influencing factor on our tool development is that we do not have a permanent development staff and are mostly relying on students for the development work. In some way or the other, most of the challenges are related to this factor:

*(C1) It is not easy to find students.* As members of a large faculty with 22 chairs, that we compete for the best students. The subjects of lectures, practical courses, and theses we are offering may look unattractive to many students compared to other offerings. Courses such as "Game development for the iPhone" will always have a greater appeal to students.

*(C2) The duration of student theses is limited.* The duration of a normal student thesis is limited to 6 months. That time frame imposes a natural boundary for the size of the work package to be worked on by the student. Furthermore, the relatively short time that we normally have with one student implies a high turnover in our student team.

*(C3) Kind of student work.* A topic for a student thesis must contain a certain amount of conceptual and theoretical work. We cannot formulate a thesis topic that only contains programming work. Yet, some of the features that we would like to see implemented consist mostly of programming work. For such features to be realized, we either have to pick special course formats with less emphasis on theoretical work than a Master's thesis or we have to tailor the topic so that it does meet the difficulty requirements.

*(C4) Qualification and suitability of students.* It is hard to judge beforehand how capable and suited a student is. Furthermore it is very difficult to not accept a student interested in a subject – firstly because we have to be thankful for every student (see C1) and secondly because of university regulations (we do not have legal means to turn down a student).

*(C5) Result quality & integration maturity.* There is a large variation in the quality of the outcomes of students' works. Some results we can use and incorporate into our tools – some we cannot. If the outcomes meet the expectations, there is usually some integration work left to be done. In the case of students struggling with our expectations, we often have more effort with supervision than for other students while the outcome may turn out not to be usable.

*(C6) Differing goals of researchers & students.* Researchers and students do not necessarily have congruent goals. Researchers want to check and validate their scientific ideas and to establish good relations with industrial partners for possible funding. Students usually want to get good marks with the least effort. Additionally – as studying is expensive – many students are interested in a job. Taking a job in industry is often more attractive as the payment usually higher.

*(C7) Development Roadmap.* It is almost impossible to establish a meaningful development roadmap as we often do not know if, when, and for how long students will be available. Scheduling detailed work packages only works *if* students are available. Any project plan is meaningless without development resources that we can reliably plan with. Also, if we find a student willing to work in our team, it does depend to some extent on the preferences of the student which work package he or she takes over.

*(C8) Implementation Technology.* For historical reasons we are using .NET for all our tools (Section 2). In the introductory lectures at our faculty, Java is used to teach students programming. Many students do not have the motivation to learn additional platforms. The chosen technology limits the number of students we can consider for our work.

*3.2. Measures*

Over time and thorugh experience, we have established measures to face the challenges listed above. In the following we describe how we handle the challenges. For each of the measures, we indicate in brackets the addressed challenge(s).

*(M1) Advertisement of courses and seminars.* We are still looking for the best ways to recruit students (C1). All our open student thesis topics, courses and seminars are held in a database on the chair's servers. The database is accessible through the chair's Web site and the theses are also displayed on the individual researchers' Web sites. Other chairs have similar facilities. In addition, courses and seminars are managed in a faculty-wide database, so there is one central site where students can find and pick their courses. We will soon move our thesis offerings to a faculty-wide database, too. Some thesis topics do not find interested students for a couple of years – sometimes until the topics do no longer fit our own strategy. Our hope is that this move will result in a greater visibility of our "job offerings". Another measure that we are experimenting with is the advertisement of courses using offline media, especially posters. For the future we are considering to post an announcement for our courses in a popular student Web forum.

*(M2) Payment for students.* A central goal is to reduce student turnover (C1, C2) and to create longer-term relationships with students. We are employing students that have proven their capabilities to keep them in the team beyond the regular thesis runtime. Contrary to an employment in industry where the students may get higher wages, the students remain very flexible when working for the research group. Work can normally be done at home and we do not require fixed working times. We are also making sure that the work does not interfere with other obligations such as learning for an exam (C6). Another advantage is that we can assign "normal" programming work to employed students – something we cannot in the context of theses (C3).

*(M3) Publications for students.* In addition to payment, we are trying to give further incentives (C2) by publishing interesting results of student works – for example from a thesis, as shown in [26] – together with the students. Some benefits for the students are (1) the publications can be helpful if the student eventually pursues an academic career, (2) it is a good exercise for the final master's thesis, (3) the students get insight and background information on the work subjects (C6).

*(M4) Project responsibility for students.* With the tools and the code base we have built over the years, it is the exception that a completely new and isolated tool is developed in student projects. Most of the practical courses or theses that we are offering deal with the implementation of new features for existing tools. The strategy and requirements come from the research staff. However, the responsibility for the code itself is usually with an employed student (C2, C6) of the core student team (see Section 4). These students are also responsible for much of the integration work (C5, C7).

*(M5) One central repository.* We have one subversion repository. The research staff and the students all work on the same repository. Students even use the repository to manage the LaTeX source files of their theses. The central repository helps to create an atmosphere of openness. More importantly, it has grown to a huge knowledge base over time. The source code, LaTeX sources and other documents can be used by everybody looking for examples or best practices (C5). One repository for all our students and for ourselves creates a certain amount of management overhead and is not common practice at our faculty. Most researchers and students maintain their own, private repositories.

*(M6) One technological platform.* We have made the strategic decision to only use C# and .NET as technological platform. Whenever a problem involves programming work, it must be done in C#. This is one of the few non-negotiable preconditions before we accept a student. The result of the strategy is that (1) we are able to provide detailed programming support to students (C4), (2) we are able to provide students with the necessary tools, for example Visual Studio, (3) we are able to understand every detail of the outcomes, (4) students can understand each others code and profit from each other. If there is a lack of resources in one tool project, it can be compensated by other members, and (5) we can integrate outcomes of different student projects into a larger software (C5). Some portions of the source code were originally developed to solve a particular problem and only much later assembled to form an integrated solution.

*(M7) Active promotion of our software.* We are actively looking for collaboration partners to use our tools and to drive development forward. (1) The tools are used in industrial cooperation projects and often demonstrated and showcased to interested parties. (2) Large parts of our code are put up on CodePlex[4]. (3) We work together with another university where we act as client and a student team at the other university was building a component for PET in a practical course (C7).

*(M8) Flexible and "opportunistic" strategy.* As mentioned above, one challenge in academic tool development is that we cannot reliably plan our development resources and we can judge the quality of the outcomes of students' works only after the fact. Our product strategy is therefore very flexible and opportunity-driven (C7). Some of the tools have not been build with a clear roadmap in mind but only after, for example, a student thesis was completed (C6). The opposite is also true where we had high expectations for a thesis but the outcomes were not usable and were not integrated into our code base. Another factor influencing our tool strategy is that we have to quickly respond to the requirements coming from our (funding) industrial partners.

*(M9) Lectures.* With regards to our chosen implementation technology one can roughly distinguish two types of students: Some students already have knowledge with .NET and are interested in the platform. Because Java is the de-facto standard in teaching at our faculty, such students are rare. There are also students who show interest in the platform but do not have any experience with it (C8). To make students familiar with .NET and to identify the interested students we are offering specialized lectures – for example a seminar about the basics of C#.

*Summary*

Summing up, we are trying to tackle the challenges in academic tool development by creating a positive working environment for students that lasts longer than just for the time of a practical course or thesis. Because the tool development is driven by industrial cooperations or our own research, the students find challenging, relevant, state-of-the-art and "real" problems to work on.

Figure 7 shows the number and distribution of student works, their types and the associated tool projects. To give an idea for the topics students work on, we refer to [24] and [25]. As can be seen in the diagram "Projects", PET is the oldest and still most active project. Especially with regards to project execution support, we are developing many new prototypes in the context of PET. The diagram "Types" shows the distribution of different kinds of student works. It (still)

[4] `http://www.codeplex.com`

***Project/Work Types Distribution***

| Project | BT | MT | DT | PC | Sum |
|---|---|---|---|---|---|
| PET | 1 | 3 | 2 | 3 | 9 |
| PDE | 1 | 2 | 0 | 0 | 3 |
| Other | 1 | 0 | 1 | 2 | 4 |
| Sum | 3 | 5 | 3 | 5 | 16 |

BT: Bachelor's Thesis
MT: Master's Thesis
DT: Diploma Thesis
PC: Practical Course

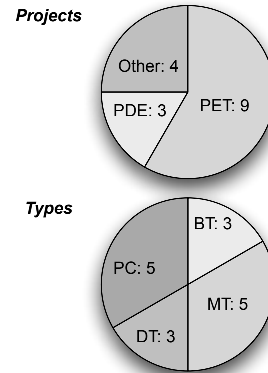



Figure 7: Distribution of projects and students' works

shows a high number of Diploma theses, which will decrease due to the *Bologna Process*[5] [5]. This development is actually an advantage for us because a student joining our group for his Bachelor's thesis usually stays at university for another two years to earn his Master's degree. Such students are candidates for employment in our group. If we get to know a student while he completes his Diploma thesis, it is too late to think about employment. Most of those candidates are leaving university soon after.

*Measures that worked.* We have observed that a new lecture is not very popular when offered for the first time. Offering the same lecture periodically, we notice increasing visibility and popularity. As an example, we are offering a seminar about object-oriented programming with C# where we have twice the number of participants in the current winter term 2010 compared to the previous year. Another example is a practical course that we are offering every semester where we have a stable number of participants.

*Measures that did not work.* For some courses, we have designed posters that we put up near the entrances to our faculty building. The first time we tried posters, these did not seem to have any effect. We had zero students for the advertised course. We are currently trying posters again for two courses and have yet to see if they make a difference to find students. So far, it appears that again the posters have no effect. If we are getting feedback from students about our lecture offerings, it is usually from students coming to us because they are interested. Much more interesting of course would be why a seminar or practical course did not generate the desired interest. We are however not aware of a working method to collect feedback from students not interested in our offerings. One possibility may be to open an "advertisement" thread in a popular student Web forum (see also measure M1). We have however not made such a posting yet.

[5] The Bologna Process harmonizes the European graduation system. In Germany, the Diploma degree is getting replaced by Bachelor's and Master's degrees.

## 4. Organization & Development Process

In this section we describe how our group is organized today. The organization has steadily evolved over the last couple of years and is basically a result of our experiences outlined in Section 3. It also reflects the fact that as members of a university, most of our funding comes from collaborations with industrial partners. Not just for the tools, the partners take on the role of the customers. How they are involved and how we work together internally with researchers and students is outlined in the following.

*Organization.* The rough structure of our research group is displayed in Figure 8. We are orga-

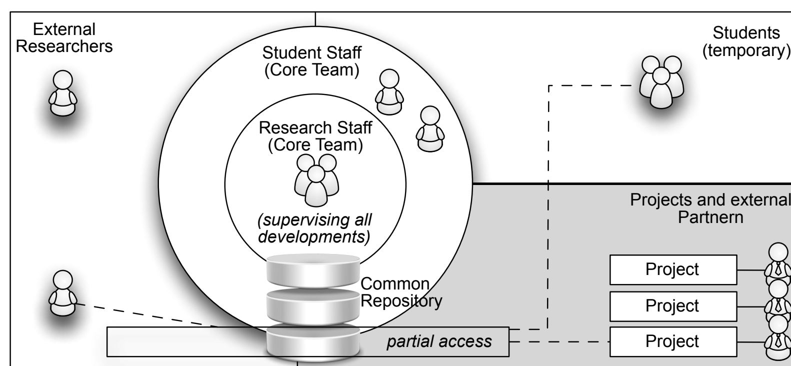



Figure 8: Organization of the research group

nized in four areas: (1) the *core team* consists of the employed researchers (including ourselves) and students. The research staff owns the *common repository*, supervises the students and coordinates projects and tool development. Each of the employed students is responsible for one tool and coordinates integration, especially of the parts developed by temporary students. (2) *External researchers* are people that do their research at our chair but are not on the chair's payroll. They are available only temporarily and are generally not as much involved as the employed staff – they do not for example have teaching obligations. (3) *Temporary working students* – e.g. working on their theses – are also embedded in the research group. (4) *External partners* participate according to their projects' context, mainly by consuming tools.

A central component is the *common repository*. All data of the group is stored there. Everybody has complete access, except for a couple of directories that contain sensitive/confidential data, e.g. grades or financial calculations. For legal and privacy reasons we cannot give access to this data to all users of the repository. These areas are only open for the employed research staff.

*Processes.* Our development process varies depending on the tool project. If it is (1) an experimental (component of a) tool, then the development is self-organized by the developing student. If it is (2) a tool that is mature for release or already released, an organized development process is established in the research group as shown in Figure 9. In general two aspects must be mentioned: At first the continuous development in the research group and furthermore developments and publications to the community. For the internal development, a team is constituted, which

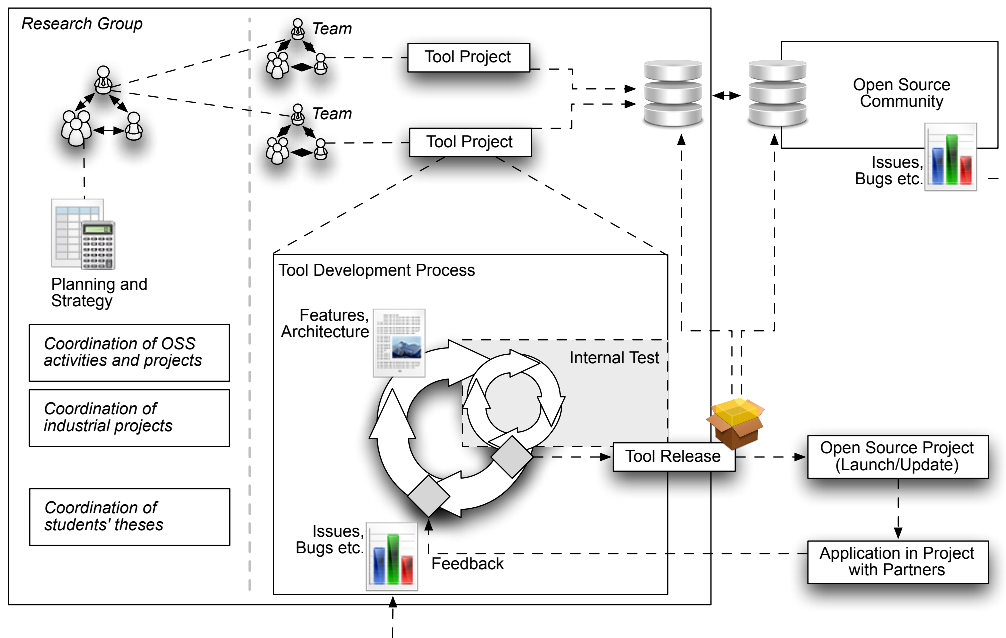



Figure 9: The tool development process of the research group

consist of at least two researchers (core team), one student (core team, *senior student*), and additional temporary students. External researchers and partners contribute where appropriate. In a *tool project*, the senior student is responsible for the development tasks in the project (coordination, integration, test). The development is planned using simple feature lists – if a tool is already published to the Open Source community, the facilities of the Open Source community site – for example issue trackers – are used.

Development starts with selecting features, defining the goals, the design of the architecture, and a rough schedule. Once the first implementation is available, the internal tests start. Internal test does not only mean running test cases but also checking the implementation against the requirements and improving the source code. Having finished the internal tests, a milestone for a (new) release is reached. At this point, two activities are performed: (1) the internal release is built and stored as a baseline in the common repository; (2) the tested code and release (executables, documentation, etc.) are published at the Open Source site. Furthermore, the published sources and tools are provided to the partners for their work. In the meantime feedback loops are established. We collect feedback from both the community and the partners. In combination with the original feature list, we plan new releases.

At this point it is important to know that the development is still driven by the research group – not by the community. The community sponsors new or updated components, but testing, integration in a release and so on is controlled by the group. The reason is quite simple: we involve temporary students who obtain their grades by taking part in our tool development. To align the strategy and the students' goals, we must control the development. This strategy creates additional effort, but is necessary to hold the working environment stable for the students.

## 5. Transfer Strategy

Academic tools need validation by practice. As most of our developments are motivated by practical problems, we always look for cooperations in industry and the research area to apply the tools. This section gives an overview over our transfer strategy.

*Academic Research & Projects.* The first part of the transfer strategy is in the academic area. We (1) use our tools for teaching. Lectures and seminars for instance include PDE besides standard environments such as the Eclipse Process Framework [4]. Practical student works as well as theses address PET. We (2) work together with other universities, e.g., the University of Applied Sciences in Rosenheim, to drive development forward (a student team of the University of Applied Sciences in Rosenheim has developed the SPEM process provider for PET – we only contributed the requirements). Furthermore, our tools are a matter of current research. We (3) use PDE and PET for modeling and proof of concepts developed in Ph.D. theses.

*Industrial Cooperation Projects.* PET has been in use for the generation of work product templates in cooperation projects for example with the Federal Network Agency and Siemens. The Federal Network Agency's internal software developers are using Microsoft Team Foundation Server. We are currently working on a process enactment strategy with the Agency involving their TFS infrastructure and PET. PDE is relatively new and early versions are delivered to selected partners. First cooperations are already established, e.g., with T-Systems where PDE is used to model the proprietary development process of this company.

Within cooperation projects, we are using the tools ourselves to gather experiences before providing the tools to partners. For example as PDE now provides the required minimum of stability and enough features for efficient work, it replaces other design tools such as EPF or the V-Modell XT reference editor [6] in our own work.

*Open Source Strategy.* To reach a broad audience, we make our tools accessible to the Open Source community. Due to our .NET framework-centered development activities we publish and manage our projects[6] on CodePlex, Microsoft's Open Source project hosting site.

Our Open Source strategy offers advantages for both the tools' users and our team. The users get to review the program's functionality and quality before using it in their projects. Through the projects' openness and availability, we hope for a growing user base that reports bugs and inspires new features, which helps us improve the tools.

As described in Section 2, our tools are specialized for a specific domain. They are no mass-market consumer tools and require a lot of know-how. The industrial cooperation or funded research projects do usually not contain a budget for tool development. As a university we are "selling" know-how – our tools are something that we bring in for free. As such, we do not lose anything by publishing the source code.

## 6. Conclusion

For our specialized work on software development processes, we have a constant demand for tool support. Many of the requirements that we have or simply ideas that we want to try out are not covered by existing tools. We are therefore developing many tools ourselves. Several mature

[6] Usually we publish the tools under the Apache 2.0 license.

tools have been developed over time. Developing the tools was possible only by finding an organization structure that respects the particular concerns (listed in Section 3) of the academic environment: we are facing some specific challenges developing tools because there usually is no budget for software development and because we do not have a dedicated development staff. Most of the development work is done by students, for example in the context of a Bachelor's or Master's thesis. We have taken certain measures and found out things that worked and things that did not with regards to our kind of "development team".

*Discussion.* Experiments such as using offline media to advertise courses have had very limited success so far. Whether moving all our thesis offerings to a central database will make a difference has yet to be seen. A measure that worked is the employment of students to build a stable team. If the students are integrated in the team, their performance often goes beyond the curriculum's requirements. So, if for example a Master's thesis is calculated with an effort of 6 man-months, the real amount of work done by the student is often two to three times as much (estimated based on the size and volume of the deliveries). *However*, this requires a special, highly self-motivated, open-minded and creative kind of students.

It is not possible to employ every student due to project, administration, and financial limitations. The selection process for students' employment must respect long-term perspectives and enough time for learning for the students as well as for the researchers' goals.

Referring to the challenges that we are facing (Section 3), our experiences lead to further questions:

- *Learning vs. working:* Students need time for their studies. We have established a structure for our employed students that enables them to control their working style. For students that work with us only temporarily, we set clearer goals – normally aligned to the curriculum's requirements. We currently have only a handful of lecture formats to choose from. A question we do not have an answer to is what adequate lecture format is for our requirements.
- *Lecture format:* Currently we only have lectures, seminars, and practical courses (1) to advertise and (2) to provide students with work. Are there other forms of teaching possible that make our area of work more attractive to students?
- *Cooperation style:* For the core team (shown in Figure 8), we have a flat organization and cooperation style where each member has certain responsibilities. The flat hierarchy creates an open working atmosphere but limits the size of the team. This style works good for us but implies a risk: It works as long as the team is relative stable, as a high turnover in the student staff results in management effort (training, repository, contracts). What other organization styles are possible?

At the same time, budgets for tool development (including students' payments) are scarce. So, we are forced also to include non-employed students with the effects stated in Section 3. The availability of non-employed students is almost impossible to plan, which destabilizes our planning and strategy. Relevant questions arise from this situation are:

- *Alternative transfer strategies:* We selected Open Source as basic concept for transferring our tools. In combination with research and consulting projects, we attract funding but must align the strategy according to the current projects. What are other strategies for a transfer? Are there alternatives to a spin-off?

- *Planning:* Planning the development depends on project situations and resources. While we do have a long-term "product strategy", it is hard to set milestones and completion dates (see Section 3). What other styles of project-independent planning exist?
- *Target achievement:* The achievement of the objectives is also closely coupled with the current projects. How well is the target achievement in other than our situations? What about the quality of the resulting tools?

The tools that have been developed in our group are evidence enough that the organization of our group and the measures we are taking to deal with the special challenges of the academic environment work in principle. The organization, the measures and the development process that we have outlined in Section 4 are the result of an ongoing learning process. Especially the development process we have not designed from scratch but refined based on the lessons learned. It has now stabilized to a point where for the first time we can consider to formalize it. Such formalization would allow us to apply PDE and PET more extensively for our own group.

Currently we have a relatively stable team that works together efficiently. However, some of our core team students have stayed with us for a couple of years and will finish studies soon. They are still the first generation of students that we have worked with in the described way. We will see how robust our organization is when we bring up a new generation. We are offering several seminars and lectures to get in touch with potential candidates in the current winter term 2010.

## References


[1] Amelunxen, C., Rötschke, T., Schürr, A., 9 2005. Graph transformations with mof 2.0. In: Giese, H., Zündorf, A. (Eds.), Proc. 3rd International Fujaba Days 2005. Vol. tr-ri-05-259. Universität Paderborn, pp. 25–31.
[2] Beck, K., 2003. Extreme Programming. Addison-Wesley.
[3] Cook, S., Jones, G., Kent, S., Wills, A. C., 2007. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley.
[4] Eclipse Foundation, 2010. Eclipse Process Framework (EPF). Online, `http://www.eclipse.org/epf`.
[5] European Commission, 2010. Focus on Higher Education in Europe 2010 – The Impact of the Bologna Process. Resource page: http://eacea.ec.europa.eu/education/eurydice/documents/thematic_reports/122EN.pdf.
[6] Federal Ministery of the Interior, 2010. V-Modell XT Online Portal. Online. URL `http://www.v-modell-xt.de/`
[7] Friedrich, J., Hammerschall, U., Kuhrmann, M., Sihling, M., 2009. Das V-Modell XT, 2nd Edition. Springer.
[8] Goodman, A., 2005. Defining and Deploying Software Processes. Auerbach Publishers Inc.
[9] Greenfield, J., Short, K., 2004. Software Factories. No. ISBN: 978-0471202844. Wiley & Sons.
[10] Hermannsdörfer, M., 2010. Migrating UML Activity Models with COPE. In: TTC 2010.
[11] Hermannsdörfer, M., Ratiu, D., 2009. Limitations of Automating Model Migration in Response to Metamodel Adaptation. In: Proceedings of Joint ModSE-MCCM Workshop on Models and Evolution.
[12] Jones, L. G., Soule, A. L., 2002. Software Process Improvement and Product Line Practice: Capability Maturity Model Integration (CMMI) and the Framework for Software Product Line Practice. Tech. Rep. CMU/SEI-2002-TN-012, Software Engineering Institute.
[13] Kneuper, R., 2008. CMMI: Improving Software and Systems Development Processes Using Capability Maturity Model Integration (CMMI-Dev), 1st Edition. No. ISBN: 978-3898643733. Rocky Nook.
[14] Kruchten, P., 2003. The Rational Unified Process: An Introduction, 3rd Edition. Addison-Wesley Longman.
[15] Kuhrmann, M., 2008. Konstruktion modularer Vorgehensmodelle. Ph.D. thesis, Technische Universität München.
[16] Kuhrmann, M., Kalus, G., Chroust, G., 2009. Tool-Support for Software Development Process. No. ISBN: 978-1-60556-856-7 in Business Science Reference. IGI Global, Ch. 11, pp. 213–231.
[17] Kuhrmann, M., Kalus, G., Then, M., 2010. Flexible Process-Tool-Integration. Research Report TUM I-1005, Technische Universität München.
[18] Microsoft Corporation (Ed.), 2007. Team Development with Visual Studio Team Foundation Server. No. ISBN-13: 978-0735625716. Microsoft Press.

[19] OMG, 2005. Software process engineering metamodel specification. Tech. rep., Object Management Group. URL `http://www.omg.org/technology/documents/formal/spem.htm`
[20] OMG, 2006. Meta object facility (mof) core specification –version 2.0. Meta object facility (mof) core specification, Object Management Group.
[21] Rombach, D., 2005. Integrated Software Process and Product Lines. In: Li, M., Boehm, B., Osterweil, L. J. (Eds.), Unifying the Software Process Spectrum, International Software Process Workshop, SPW 2005, Beijing, China, May 25-27. Lecture Notes in Computer Science. Springer, pp. 83–90.
[22] Sych, O., 2007. T4: Text template transformation toolkit. http://www.olegsych.com/2007/12/ text-template-transformation-toolkit/.
[23] Ternité, T., Kuhrmann, M., 2009. Das V-Modell XT 1.3 Metamodell. Research Report TUM-I0905, Technische Universität München.
[24] Then, M., Februar 2010. Design of an intelligentwork item tracking system. Bachelor Thesis.
[25] Wachtel, E., 2010. Design of a Domain Specific Language for designing of process models based on the V-Modell XT Metamodell. Master's thesis, Technische Universität München.
[26] Wachtel, E., Kuhrmann, M., Kalus, G., 2009. A Domain Specific Language for Project Execution Models. In: Fischer, S., Maehle, E., Reischuk, R. (Eds.), 39th Annual Conference of the German Computer Society. Lecture Notes in Informatics (LNI). pp. 2986–3000.
[27] Webb, J., 2007. Essential SharePoint 2007: A Practical Guide for Users, Administrators and Developers, 2nd Edition. O'Reilly Media, iSBN: 978-0596514075.

# A Platform for Experimenting with Language Constructs for Modularizing Crosscutting Concerns

Tim Molderez, Hans Schippers, Dirk Janssens

*Antwerp Systems and Software Modeling (Ansymo)*

*University of Antwerp, Belgium*

Michael Haupt, Robert Hirschfeld

*Software Architecture Group*

*Hasso Plattner Institute, Potsdam, Germany*

**Abstract**

When implementing a new programming language construct, it is important to consider and understand its implications on program semantics. Simply hacking compiler code, even in combination with the use of a debugger, does not allow for easily keeping track of the global picture of overall execution semantics. We present a graph-based implementation of the delMDSOC virtual machine (VM) model in AGG, as a platform for experimenting with programming language constructs. More specifically, given the nature of the delMDSOC model, it is aimed primarily at languages supporting the modularization of crosscutting concerns, such as aspect-oriented or context-oriented languages. Our delMDSOC implementation visualizes programs as graphs at the VM level, in terms of well-known and intuitive concepts: objects, messages, delegation and actors. Implementing new high-level language constructs involves expressing them in terms of these concepts. Since delMDSOC is implemented as a graph rewriting system, program execution can be visually simulated and program state can be inspected at all times, providing insight in the implications of the new language construct on execution semantics. We demonstrate our approach by means of two language constructs: the context-oriented layer construct and a new aspect-oriented construct, the "concurrent cflow" pointcut.

*Keywords:* virtual machine model, programming language development, modularizing crosscutting concerns, graph rewriting

## 1. Introduction

New constructs are being added to programming languages regularly, sometimes leading to language extensions or even completely new programming languages. For the language devel-

*Email addresses:* tim.molderez@ua.ac.be (Tim Molderez), hans.schippers@ua.ac.be (Hans Schippers), dirk.janssens@ua.ac.be (Dirk Janssens), michael.haupt@hpi.uni-potsdam.de (Michael Haupt), hirschfeld@hpi.uni-potsdam.de (Robert Hirschfeld)

oper, as well as the language user, it is crucial to understand the semantics of such new constructs, and their implications on the execution semantics of programs written in this language.

New constructs are often simply "hacked in" by modifying compiler or interpreter software, the semantics being established by their implementation. The language user then either relies on high-level documentation, which is often lacking, or on the compiler or interpreter itself, using it as a black box to figure out the semantics of such constructs.

This paper presents a platform that can be used to experiment with object-oriented language constructs. This platform is based on the delMDSOC (delegation-based Multi-Dimensional Separation Of Concerns) virtual machine (VM) model [1] and its implementation in the AGG graph transformation tool [2]. In essence, delMDSOC is a VM model that is designed as a compilation target for programming languages aiming to increase modularization and enhance (multi-dimensional) separation of concerns (MDSOC). Examples of such languages include aspect-oriented programming [3], context-oriented programming [4] and role-based programming [5]. Much attention and effort is devoted to experimenting with new language constructs within this fairly young area of programming language research [6, 7, 8, 9].

Our AGG implementation of delMDSOC visualizes programs as graphs at the VM level. The behavior of the programs represented by these graphs is determined by a set of graph rewrite rules [10] which constitute the operational semantics of delMDSOC. As such, AGG allows for visual simulation of program execution by repeatedly transforming the program graph through application of these rules. In between rule applications, execution can be paused and the entire program state can be inspected and modified, including the VM code that each object implements, and even the runtime stack. Because program state is represented by a graph, it also is possible to make backup copies thereof, such that execution can always be resumed from a backup.

What makes this platform useful for experimentation is the fact that, unlike most intermediate languages, delMDSOC's abstraction level is much closer to a high-level programming language than it is to a typical machine instruction set. Because the distance between the programming language and the VM instruction set is decreased, it becomes easier to reason about the programming language. More concretely, languages that are implemented on top of delMDSOC will be expressed in terms of the following well-known high-level concepts: prototype objects, message sending, delegation and actors. A wide range of language constructs can be created using these building blocks [1, 11], due to the delMDSOC model's dynamic nature: All message sends are late-bound and delegation relations between objects can be modified at runtime.

The remainder of this paper is structured as follows: Sec. 2 provides an overview of the delMDSOC machine model. Next, Sec. 3 presents the model's operational semantics in depth. In Sec. 4, we apply our semantics to two examples of language constructs: the existing context-oriented layer construct and a new aspect-oriented construct, the "concurrent cflow" pointcut. Sec. 5 discusses related work, and 6 concludes and briefly outlines future work.

## 2. Overview of the delMDSOC model

The delMDSOC machine model was originally introduced in [1]. In this section, we give an overview of the model's operation.

As the model is a prototype-based object-oriented environment, the model's basic entities are objects, which can communicate with each other by means of message passing. We opted for prototype objects in favor of a class-based object-oriented model because we wanted the model

Figure 1: An example of a composite object

to consist of a minimal amount of concepts, keeping it as simple as possible. As there is no concept of classes in prototype-based languages, this was an easy choice to make.[1]

Objects can be connected to each other using delegation, resulting in a chain of objects, also called a *composite object*. An example of such a composite object is shown in Fig. 1. During message lookup, if an object receives a message it cannot understand, the message is delegated to the next object in the chain, and so on, until a matching implementation is encountered in one of the objects' message dictionaries.

In delMDSOC, the first object in a composite object's delegation chain is a so-called *proxy*, which does not understand any messages at all. Its purpose is to serve as a fixed access point, establishing the identity of the composite object. This is important because delegation chains are not fixed, but may be modified at runtime. This mechanism allows for dealing with dynamic deployment of crosscutting modules. In such a scenario, objects may be inserted into or removed from delegation chains in order to dynamically modify a composite object's behavior in response to certain messages. The `listCache` object in Fig. 1 is an example of a crosscutting module that is currently enabled, as it is inserted between the `list` proxy and the `listObj` object. During message lookup, the *self* pseudovariable always remains bound to the receiver of the message, i.e., the proxy, ensuring that lookup for messages sent by a composite object to itself always starts at the front of the delegation chain.

In order to introduce concurrency into delMDSOC, we have opted for the actor-based model of concurrency. Actors were chosen in favor of threads, also because of simplicity reasons: Because actors do not allow for shared state, issues such as deadlocking are avoided for the most part. However, we do not adhere to the pure actor model, but rather take an approach similar to the one employed in the E language [12], in which a distinction is made between actors and objects: Actors act as containers of objects. An object can send an asynchronous message to an object belonging to another actor, which results in this message being appended to the other actor's *mail queue*. The mail queue essentially is a buffer where messages are kept until the message that is currently being processed has finished its execution. When this happens, the message at the front of the mail queue is removed and is pushed onto the *process stack*. Although an actor may at all times receive additional messages in its mail queue, it will not process them until it has dealt with the messages currently residing on the process stack. Furthermore, executing a message on the process stack may result in other message sends. If these messages target objects within the same actor, they immediately end up on the process stack, which means they are processed synchronously. In short, communication between objects within one actor (intra-actor communication) occurs synchronously, whereas communication involving two different actors (inter-actor communication) occurs asynchronously by default. Asynchronous inter-actor communication may result in a *future* object [13] being returned. Upon accessing a future's value, the actor is blocked until the value has been calculated. This blocking mechanism can be also used in order to simulate synchronous inter-actor communication.

[1] If needed, classes can always be emulated using prototype objects in combination with delegation [1].

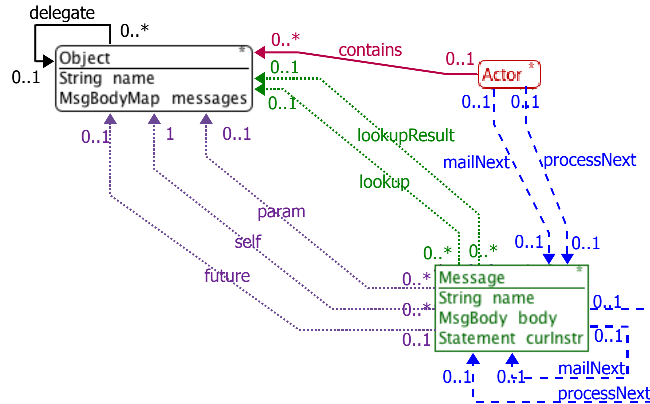



Figure 2: Type graph of the delMDSOC model

### 2.1. *The model's operation*

The delMDSOC model is described as a set of graph rewrite rules. These rules can be subdivided according to the cycle of steps that each actor goes through:

1. Move the first message from the mail queue to the process stack. (This step is detailed in Sec. 3.1.)
2. Perform the lookup procedure to find the message's implementation, also called the message body. (Sec. 3.2)
3. While the message body has not been executed completely:
   (a) Fetch the next instruction from the message body.
   (b) Perform some preprocessing on this instruction, if any. (Sec. 3.3)
   (c) Execute the instruction. (Sec. 3.4)
4. Pop the message from the process stack and start over at step 1.

While the model's graph rewrite rules can be matched in a non-deterministic order, they are set up such that each actor will indeed follow the above sequence of steps. It should also be mentioned that, while delMDSOC supports a whole set of different instructions, this paper will only focus on a few of these instructions. Several instructions, such as integer addition, do not modify the graph structure, but only attribute values, which is of limited interest in this context. Instead, message sends and object (un)deployment will be discussed in depth in Sec. 3.4. These are more interesting because the mapping of several high-level MDSOC constructs onto the delMDSOC machine model will involve these instructions.

### 2.2. *Type graph*

Fig. 2 shows the type graph associated with our model, which specifies the model's different kinds of nodes and edges, and how both connect to each other. Three kinds of nodes are available: actors, objects and messages. Node attributes are typed by Java classes.[2] Object nodes contain

[2]It is important to note that we only make use of a minimal subset of Java's functionality. Java expressions are only used in order to read and update attribute values, but they are not allowed to cause side-effects to the graph structure.

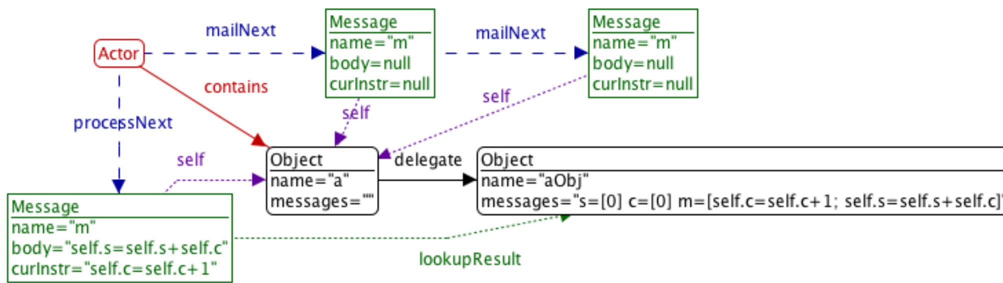



Figure 3: An intermediate state of a simple program

two attributes: `name` and `messages`. The `name` attribute is a `String` acting as a unique identifier. The `messages` attribute is a message dictionary containing implementations for the messages it understands. It has the `MsgBodyMap` type, which is a wrapper class around a hash map, mapping each message name to a list of instructions.

Message nodes contain three attributes: `name`, `curInstr` and `body`. Once a message has been looked up, the message body is stored in the `body` attribute as a `MsgBody`, a linked list of `Statements`, the latter being a wrapper around a `String`. When a statement is executed, it is removed from `body` and is then stored in the `curInstr` attribute, which represents the instruction currently being executed.

Next to the three types of nodes, there also are several types of edges:

- The `contains` edge connects an object to the actor containing it.
- The `delegate` edge indicates the next object in a composite object's delegation chain.
- The `mailNext` edges form an actor's mail queue. If the source node of a `mailNext` edge is an actor, the edge indicates the first message in an actor's mail queue. Otherwise, it indicates the next message in a mail queue.
- Analogous to the `mailNext` edge, the `processNext` edges are used to form an actor's process stack.
- The `lookup` edge indicates the object currently being checked in the message lookup procedure. A `lookupResult` edge then indicates the object where a message was understood.
- The `self` edge indicates a message's self object. Analogously, the `param` edge indicates a message's parameter object, used to pass parameters along with a message send.
- The `future` edge indicates that a message's return value should be stored inside a future object.

The graph in Fig. 3 illustrates an example that complies with the above type graph. It shows an intermediate state of a running program that will calculate the value of $\sum_{c=1}^{3} c$ and store the result in `aObj.s`. It incorporates several of the model's elements: There is one actor that contains one composite object `a`. The actor has two pending messages in its mail queue and is currently executing the instruction `self.c=self.c+1`, as shown in the message that is on the actor's

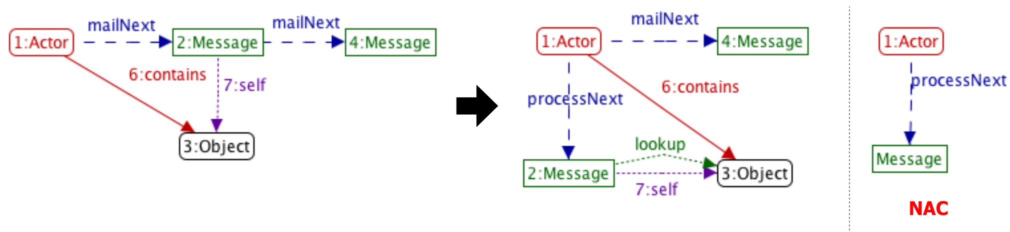



Figure 4: Process message rewrite rule

process stack. The rewrite rules described in the following section will give a general idea on how the execution of this program will continue.

## 3. Graph rewriting semantics of the delMDSOC model

This section presents the essentials of delMDSOC's semantics, in the form of a set of single-pushout graph rewrite rules.[3] Each graph rewrite rule contains at least two graphs: a left-hand-side (LHS) and a right-hand-side (RHS). The application of a rule then consists of searching for the rule's LHS within the graph that we wish to transform. If a match was found, it will be replaced by the RHS and the rule application will have succeeded; otherwise, the rule application has failed. Optionally, a rule can also have multiple attribute conditions and negative application conditions. An attribute condition (AC) is a Boolean expression that must evaluate to true in addition to matching a rule's LHS. A negative application condition (NAC) specifies a subgraph that may not occur when matching a rule.

### *3.1. Processing messages in the mail queue*

The graph rewrite rule in Fig. 4 comes into play when an actor has an empty process stack, which is ensured by the rule's NAC. If an actor's process stack is empty, this means it is waiting for new messages to process. The rule's RHS removes the first message from the mail queue and moves it onto the actor's process stack. The message also receives a `lookup` edge, indicating that the lookup procedure should start, which is handled by the rules in Sec. 3.2. It should be noted that this rule will not work in the case where only one message is in the actor's mail queue. This issue was solved by making use of AGG's amalgamated rules, which makes it easier to create, maintain and combine variations on rules, rather than having to specify each variant as a separate rule. Using this technique, a variant (not shown here) was created that handles the case with one message in the mail queue.

### *3.2. Message lookup*

The lookup procedure described in this section behaves exactly like the lookup procedure discussed in Sec. 2. In case the desired message implementation is not found in the object currently indicated by the `lookup` edge, the rule in Fig. 5 matches. This is ensured by attribute condition `!msg.contains(m)`, which checks whether the object contains an implementation

[3]The complete set of graph rewrite rules as described in AGG is available at http://fots.ua.ac.be/delmdsoc/.

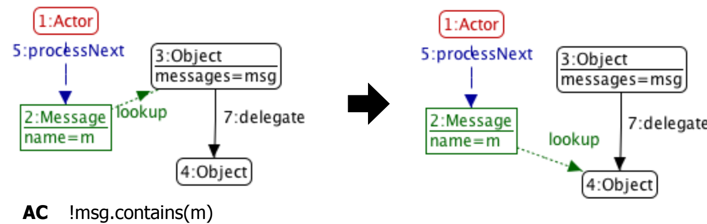



Figure 5: Iteration rule of the lookup procedure

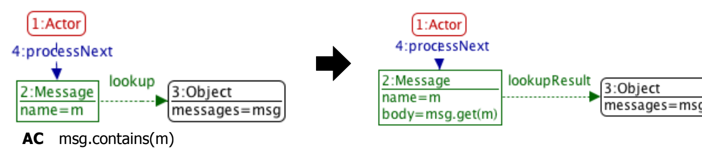



Figure 6: The end of the lookup procedure

for a message named `m`. Its RHS moves the `lookup` edge to the object's delegate, such that the lookup procedure continues there.

In the other case, if the desired implementation was found in the current object, the rule in Fig. 6 matches. Its RHS copies the implementation into the message's `body` attribute and the `lookup` edge is changed into a `lookupResult` edge, indicating that the lookup procedure has finished and that the body can now be executed.

### 3.3. *Preprocess the current instruction*

Before an instruction can be executed, some preprocessing may be required. Any references to the `self` pseudovariable in an instruction are replaced by the name of the self object, as indicated by the current message's `self` edge. Similarly, references to formal parameters and future objects can be resolved in the preprocessing phase.

Instructions that contain subexpressions that haven't been evaluated yet are also handled during preprocessing: There is a rule that will push a new message on the process stack that is meant to evaluate this subexpression. Once this is finished and the message can be popped, the subexpression's resulting value is resolved in the original instruction.

### 3.4. *Execute the current instruction*

The intermediate language implemented by our machine model supports several different types of high-level instructions:

There are the usual assignments, if statements, print statements and (implicit) return statements, but there also are more interesting instructions for sending messages, resending messages (analogous to the aspect-oriented proceed statement), creating new actors, cloning objects and (un)deploying objects in delegation chains. This section concentrates only on sending messages and object deployment, as these form the essence of the examples in Sec. 4.

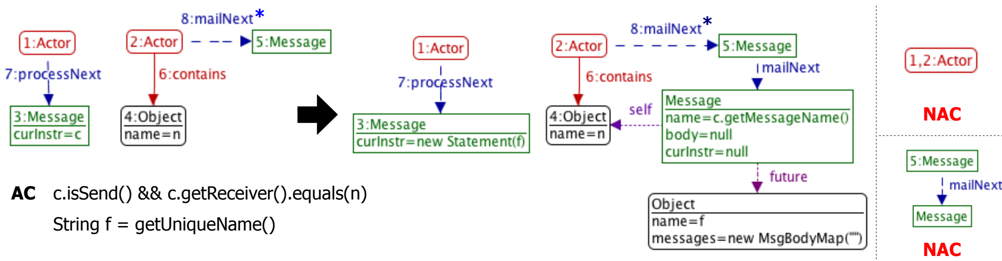



Figure 8: Inter-actor communication

*3.4.1. Intra-actor communication*

The rule handling communication between objects within the same actor, i.e., intra-actor communication, is shown in Fig. 7. The first AC makes sure the current instruction has the syntax of a message send and that its receiver equals the object with name `n`. (For example, the instruction `listObj.get(3)` sends the message `get` to receiver `listObj` with parameter `3`.) The second AC will check for the presence of the actual parameter object that is passed along with the message send. For simplicity reasons, the delMDSOC model can currently only pass one parameter; multiple parameters can be simulated by using this one parameter object as a container. It also is possible to perform a message send without any parameters; amalgamated rules are used here once again to define the variant of this rule without parameters.

Once this rule matches, the new message that will be sent is pushed onto the actor's process stack. As this message is now at the top of the stack, the other messages on this stack have to wait until the new message has been looked up and executed. In other words, this is a synchronous message send.

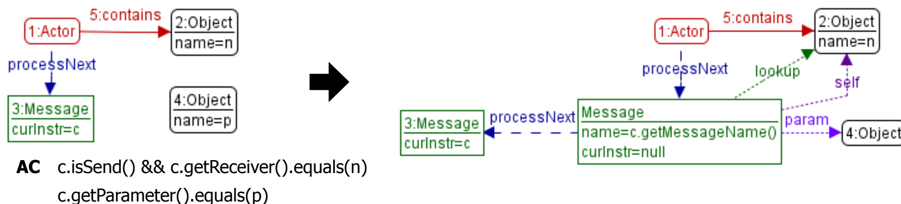



Figure 7: Intra-actor communication

*3.4.2. Inter-actor communication*

Sending messages between two different actors is handled by the rule in Fig. 8. This rule is one of the few occasions where it becomes apparent that our graph rewriting rules do not use injective matching, in which each node in a rule must be different from each other. This means it could occur that both actors, node 1 and 2, are the same actor. To prevent this situation, a NAC was added that ensures the two actor nodes may not be the same.

In contrast to intra-actor communication, the new message does not immediately end up at the top of the receiving actor's process stack, but it is added to the back of its mail queue. To

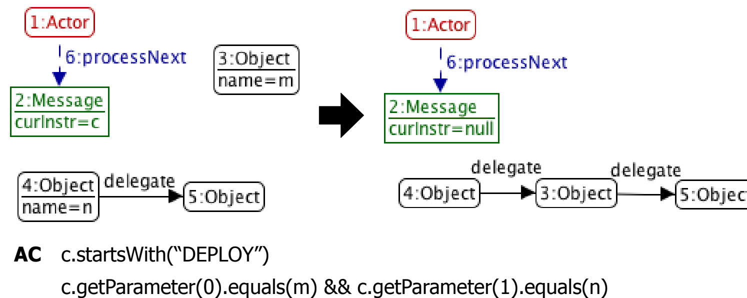



Figure 9: The deployment instruction

retrieve the last message in the mail queue, transitive closure[4] (marked by the asterisk on edge 8) is used in combination with another NAC. The actor that initiated the message send can continue immediately, meaning that message sends between different actors are asynchronous by default.

To be able to move the return value back to the sending actor once it is available, we make use of futures [13]. In the rule's RHS, a new object is created with a unique name `f` and a `future` edge pointing to it; this is a future object. Once the message that was sent has been executed, the return value will be stored inside this future object. In the meantime, the message that caused the message send (node 3) will receive a reference to the future object as a temporary return value. This reference can be assigned to fields and be passed around as a parameter without a problem, even though the future object may still be empty. When accessing the future object however, the actor will block until the return value is present in the future object. Once the return value is available, another rewrite rule will make sure that accesses to the future object are replaced by the return value contained within. All of this happens in a transparent manner, in the sense that someone making use of delMDSOC does not need to be aware of the existence of future objects to be able to use them. Synchronous inter-actor communication can be simulated using the future's blocking mechanism, which is done by making an asynchronous call and immediately attempting to access its return value. An explicit `SYNC` instruction is available that provides this functionality.

*3.4.3. Deployment*

The deploy instruction is a key component that allows the model to be used in an MDSOC context. This instruction is used to insert an object within another specified delegation chain. For example, when executing the statement "`DEPLOY(objectA,objectB)`", `objectA` will be inserted between `objectB` and its delegate. The rule handling this instruction is shown in Fig. 9. The attribute conditions of the rule express that the current instruction must be a deploy instruction and that both of its parameters should match with the objects respectively named `m` and `n`.

[4]AGG currently does not support transitive closures, so an additional type of edge is used to explicitly mark the last message in a mail queue.

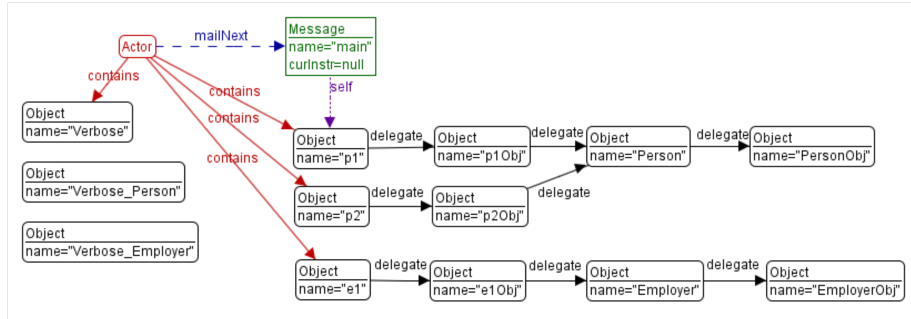



Figure 10: Initial program state

## 4. Examples

We will now, by means of two examples, illustrate how the set of graph rewrite rules described in the previous section, as well as their simulation in AGG, can be a useful tool while experimenting with language constructs. The first example involves an existing construct that forms the basis of context-oriented programming: layers. The second example introduces a new variation on the aspect-oriented cflow pointcut, called concurrent cflow, or `ccflow`.

### *4.1. The context-oriented layer construct*

Before diving into any details of the example, we will first give a brief introduction to context-oriented programming [4]. The main idea behind context-oriented programming is that some functionality may behave differently when executed in a different context. To this end, the layer construct was introduced; a layer contains the behavioral variations of certain methods in a particular context. More concretely, alternate definitions of any method can be written within a layer. These alternate definitions can then be explicitly activated, i.e. replace their original definitions, whenever a particular dynamic scope is entered.

Listing 1 shows a simple context-oriented example that we wish to map to the delMDSOC model. It is written in a ContextJ-like language [4], which is a context-oriented extension of Java. In this example, two classes are present, `Person` and `Employer`; each of these defines a `toString` method. We redefine the `toString` method of each class within the `Verbose` layer. Also note the use of the `proceed` statement within this layer definition; this represents a call to the original method definition. In `Person.main`, the layer is activated using the `with` block construct. In other words, all code executed in the dynamic scope of the `with` block will make use of the redefined `toString` methods.

Fig. 10 shows the corresponding representation of the initial state of this program in delMDSOC. (The `messages` and `body` attributes are hidden, as they would clutter the graph.) At this point, the `Verbose` layer is not active yet. The `main` method is called by sending a `main` message to object `p1`. Given that delMDSOC assumes an object-based environment, classes are represented as objects, just like their instances. Instances hold fields corresponding to the attributes of their class, and delegate to the class object, which holds a method dictionary. Recall from Sec. 2 that all objects are represented as a combination of a proxy and the actual object. Hence, all information regarding a high-level object is captured by four machine-level objects: An object representing the instance, an object representing the class and a proxy for each of these. Class

```
class Person {
    String name; Employer emp;
    String toString() {return name;}
    void main() {
        Employer e=new Employer("Cosmo Spacely",
            "Sprocketlane 23, Orbit city");
        Person p=new Person("George Jetson", e);
        with(Verbose) {
            System.out.println(p.toString());
        }
    }
}

class Employer {
    String name; String addr;
    void toString() {return name;}
}

layer Verbose {
    void Person.toString() {
        return proceed() + "Employer:" + emp.toString();
    }
    void Employer.toString() {
        return proceed() + "Address:" + addr.toString();
    }
}
```

Listing 1: Example context-oriented program

objects can be reused, which is why `p1` and `p2`, which are both instances of `Person`, partially share the same delegation chain in Fig. 10. Apart from `p1` and `p2`, there is also an `Employer` instance `e1`. The `Verbose` layer is represented by the objects `Verbose`, `Verbose_Person` and `Verbose_Employer`. The first implements the logic of layer activation and deactivation; the other two contain the layer's redefined `toString` methods. Because we are not dealing with a concurrent program, only one actor node is present, which contains all objects (by design, only objects that can receive messages need to be connected by a `contains` edge). The actor's mail queue holds a message `main`, which will be processed by removing it from the mail queue and pushing it onto the process stack, looking up a corresponding method body in the delegation chain of the receiver `p1`, and executing it.

Applying the graph rewrite rules relevant for this lookup process (cf. Sec. 3.2) will result in the following definition of `main` being found in the `Person` class object:

```
...
Verbose.activate
PRINT(p2.toString)
Verbose.deactivate
```

Essentially, the `with` construct from Listing 1 was translated into two message sends to the `Verbose` object: An `activate` message upon entrance of the scope, and `deactivate` upon exit. A definition for these two messages is indeed provided in `Verbose`'s method dictionary, where `activate` is defined as:

```
DEPLOY(Verbose_Person,Person)
DEPLOY(Verbose_Employer,Employer)
```

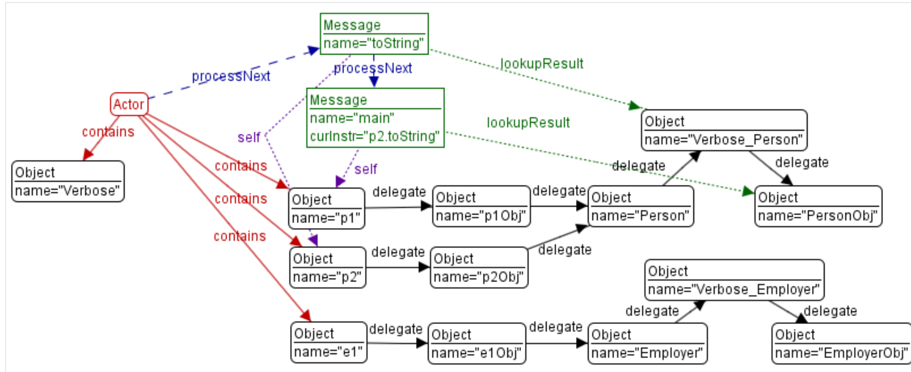



Figure 11: Program state after activation

The instructions will cause the graph rewrite rule for deployment (cf. Sec. 3.4) to be applied, resulting in the two objects being inserted in the delegation chains immediately after the object provided as the second parameter. The definition of `deactivate` simply reverses this effect:

```
UNDEPLOY(Verbose_Person)
UNDEPLOY(Verbose_Employer)
```

Fig. 11 shows the situation where the layer has already been activated, the message `toString` has been looked up in the delegation chain of `p1` and a definition was found in `Verbose_Person`, as can be inferred from the `lookupResult` edge between the message and `Verbose_Person`. The latter, just like `Verbose_Employer`, has been inserted in the delegation chain of the appropriate class object, as a consequence of the layer's activation. After executing the `toString` message, the `Verbose` layer is deactivated again and the delegation chains are restored to the state in Fig. 10.

As the graph rewrite rules defining the operational semantics of delMDSOC have been specified using the AGG tool [2], the example outlined above can be simulated automatically. This effectively visualizes the operational semantics of a context-oriented program in terms of objects, messages and delegation. From the above example we derive that classes are mapped onto normal low-level objects, as are their instances, layers are represented by a set of objects for each method they override and an object that implements the layer's activation and deactivation, respectively upon entrance and exit of the corresponding dynamic scope.

For language developers, being able to break down program execution in basic simulation steps while visualizing object relations and program state is a useful tool in order to verify whether a new language construct behaves according to desirable semantics.

For the language user, it may help in truly understanding the impact of a particular language construct or in fact a whole programming paradigm.

### 4.2. *The concurrent cflow pointcut construct*

Consider the sample code in Listing 2, written in an imaginary aspect-oriented language with support for actor-based concurrency. In this example, two actors are present, an `Admin` and a `Client`, both of which can manipulate a database, independent from each other. At any point however, the administrator may initiate a backup of the database. This can cause a synchronization issue, in the sense that updating the database while the backup is created

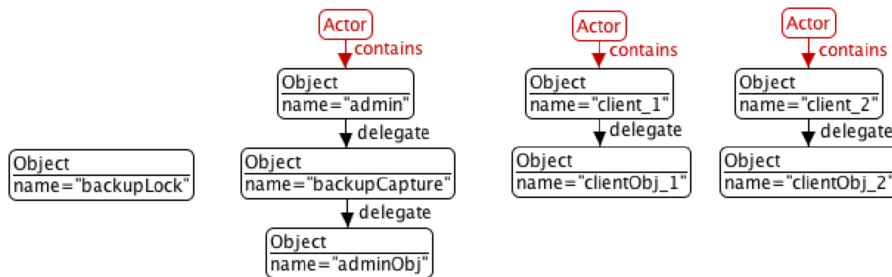



Figure 12: Initial object configuration

could render this backup file corrupt. A simple strategy to prevent this from happening is to block any attempts to update the database during the backup operation. This is expressed by the `BackupLock` aspect, which makes use of a special "concurrent cflow" construct, or `ccflow`, in its pointcut. This construct is a concurrent variant of the aspect-oriented `cflow` pointcut construct as present in AspectJ [3]. The intuition behind this aspect is that, for the duration of the `backup` call in the `Admin` actor, all calls to the `update` method in the `Client` actor should be intercepted and result in an error message. The expressiveness of the existing `cflow` construct is insufficient in this scenario, since it can only be applied within a single thread of control. Any `update` call will be in another control flow than the one of `backup`, as they are executed by different actors. In Sec. 4, we will show how delMDSOC can help in establishing the semantics of this new `ccflow` construct.

```
actor Admin {
  DbConnection conn;
  void backup() {
    conn.backup();
  }
}
```

```
actor Client {
  DbConnection conn;
  List read(...) {
    return conn.query(...);
  }
  void update(...) {
    conn.query(...);
  }
}
```

```
aspect BackupLock {
  around():ccflow(execution(Admin.backup))
  && execution(Client.update) {
    print("ERROR: access denied");
  }
}
```

Listing 2: Example usage of the `ccflow` construct

The initial object configuration of the example in its delMDSOC representation is shown in Fig. 12. The figure shows three composite objects representing the database's administrator, `admin`, and two instances of client users, `client_1` and `client_2`. The database itself and any class objects[5] are not shown here, as they do not play a significant role in this example.

The pointcut needed to capture all database updates across all clients during a backup operation is as follows, in an AspectJ-like syntax:

[5] Classes can be simulated in a prototype-based environment by letting objects representing instances delegate to an object that represents the class.

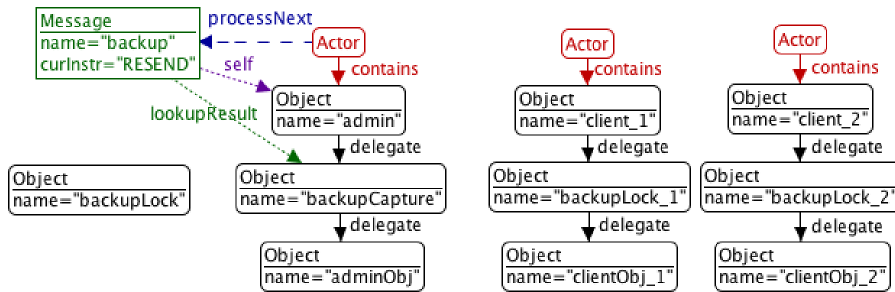



Figure 13: After deploying the `backupLock` objects

```
ccflow(execution(Admin.backup)) && execution(Client.update)
```

The `ccflow` pointcut construct is an extension of the regular `cflow` construct. It captures *all* join points across all actors in the duration of the specified control flow, whereas regular `cflow` can only capture the join points *within* a particular control flow.

To be able to capture the control flow of a backup operation, the object `backupCapture` has already been deployed in the `admin` composite object. The example scenario will be initiated once a `backup` message is sent to `admin`. This message will be intercepted by `backupCapture` before it can reach `adminObj`, which implements the backup operation itself. The implementation of the `backup` message in the `backupCapture` object can be expressed as follows:

```
SYNC(client_1.deployLock)     // Deploy backupLock at client_1
SYNC(client_2.deployLock)     // Deploy backupLock at client_2
RESEND                        // Do the backup operation
client_2.undeployLock
client_1.undeployLock
```

The first two instructions will deploy a clone of the `backupLock` object in each client's delegation chain. The `backupLock` objects will intercept any `update` message; its implementation of `update` contains our aspect's advice; it simply prints an error message stating that the database may currently not be updated. Also notice the use of the `SYNC` construct, first mentioned in Sec. 3.4.2. It indicates that the enclosed inter-actor message send should be synchronous. This ensures that our aspect is deployed *before* the backup operation is started. The program's current state after the two `backupLock` objects have been deployed is shown in Fig. 13. At this point, the actual backup operation can safely proceed. This is done with the `RESEND` statement; it resends the backup message starting at `backupCapture`'s delegate, `adminObj`. Once finished, we have left the control flow of the backup operation, so we can revert back to the original object configuration by undeploying the `backupLock` clones. All users are now free to modify the database once again.

A few remarks should be made regarding this example: Firstly, the solution presented here assumes that the `backup` operation does not contain any recursive calls. If a recursive backup call were made, it would be intercepted by the `backupCapture` object again, which is not desired. This can be solved by letting the `backupCapture` object undeploy itself before the backup operation is started and then redeploying itself once the operation has finished. A second remark about this solution is that our aspects, being the `backupLock` objects, are deployed

per instance. This allows the `backupLocks` to keep their own separate state, which could be used, for example, to maintain a separate log of attempted database updates per client. However, this comes at the price of added complexity: The fact that clients can be added and removed at runtime should also be taken into account. While not shown in this example, this is handled by intercepting the instantiation of new clients, i.e., intercepting the `new` message that is sent to the client class object. For every new client that is created, a copy of the `backupLock` is deployed into the client's delegation chain. Additionally, the `backupCapture` object is notified of every client that is added, who will maintain a list of all available clients. Whenever the backup operation finishes, `backupCapture` will use its list of clients to undeploy all `backupLock` instances.

## 5. Related work

In the field of MDSOC languages, several different semantics are available, especially for aspect-oriented languages. Typically, such semantics are expressed as a structural operational semantics [14, 15, 16, 17]. However, there are a few instances in which graph rewriting is used: In [18], a graph-based operational semantics is provided for an aspect-oriented extension of Featherweight Java, where it is used for verification purposes. The idea of a machine model as a target for a multitude of MDSOC paradigms is not present however. This is apparent, for example, in the fact that an explicit *proceed* stack is modelled, in order to accommodate for the corresponding high-level language construct.

There are also a few instances of intermediate languages taking a high-level approach: The SUIF [19] infrastructure provides a high-level object-oriented intermediate representation and a toolkit for mapping programming languages onto this intermediate representation. This infrastructure is used as a vehicle to experiment with compiler optimization techniques for languages such as C and Fortran. While not the primary focus of our model, this suggests that delMDSOC may be a useful platform to experiment with MDSOC-specific optimization techniques.

More recent work includes the C Intermediate Language (CIL) [20], which provides a high-level representation of C programs. Similar to the delMDSOC model, CIL aims to rely on a minimal amount of concepts, making it easier to reason about C programs. Rather than using CIL for experimentation purposes, it is used for analyzing program properties such as memory safety and for performing source-to-source program transformations.

## 6. Conclusion and Future Work

This paper has introduced the delMDSOC virtual machine model as a platform to experiment with language constructs. The model's graph-based semantics in combination with the AGG tool allow the language developer to simulate programs making use of the model. This simulation is well-suited for experimentation in the sense that a program's entire state is visible; any aspect of it can be freely modified whilst the simulation is paused; the program's state can also be copied and stored, such that simulation can always resume from that point. However, a scalability problem does arise due to the fact that the entire state is visible. Therefore, our current approach is only applicable to small toy examples, which may not be sufficient for experimenting with more complex language constructs. Improved graph rewriting tool support should help significantly though: There is a need for better ways of navigating large graphs, improved layouting algorithms and better means to filter out the information that currently is irrelevant to the user.

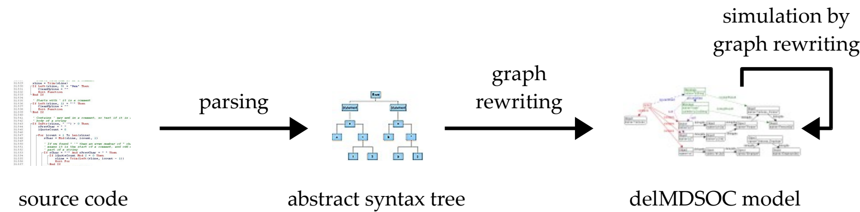



Figure 14: Constructing a new language with delMDSOC

The delMDSOC model itself is not yet considered to be in a finalized state. For example, the fact that the concurrency model is currently actor-based, is not a dogma. In fact, graph rewriting has been used before to model different approaches to concurrency, including process algebra [21] and Petri nets [22].

Regarding the experimentation platform itself, there also are several areas of future work that can be explored: Currently, the translation of programs into an AGG representation is a manual process. Ideally, this would be an automated process for each language. One approach would be to write a parser that uses the AGG API as a backend in order to generate an appropriate delMDSOC input graph. Another approach would be to have the parser only generate an AST in AGG format, and apply graph rewriting within AGG once more in order to transform the AST into an appropriate delMDSOC input graph. Fig. 14 illustrates the latter approach.

Another area of future work deals with the use of delMDSOC as a common framework in which MDSOC languages can be compared and integrated. The model may also be used for verification purposes, e.g., one can investigate the effects of the ordering of objects within a delegation chain or the condition under which it is safe to deploy or undeploy objects in remote actors.

## References

[1] M. Haupt, H. Schippers, A machine model for aspect-oriented programming, in: ECOOP 2007 - Object-oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings, Vol. 4609 of Lecture Notes in Computer Science, Springer, 2007, pp. 501—524.

[2] G. Taentzer, AGG: a graph transformation environment for modeling and validation of software, in: Applications of Graph Transformations with Industrial Relevance, 2004, pp. 453, 446.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-Oriented programming, in: M. Akşit, S. Matsuoka (Eds.), Proceedings European Conference on Object-Oriented Programming, Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, 1997, p. 220–242.

[4] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-Oriented programming, Journal of Object Technology, ETH Zürich 7 (2008) 125–151.

[5] S. Herrmann, Object teams: Improving modularity for crosscutting collaborations, in: NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, Springer-Verlag, 2003, p. 248–264.

[6] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding trace matching with free variables to AspectJ, in: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, San Diego, CA, USA, 2005, pp. 345–364.

[7] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, K. Kawauchi, Event-Specific software composition in Context-Oriented programming, in: Software Composition, 2010, pp. 50–65.

[8] A. Popovici, T. Gross, G. Alonso, Dynamic weaving for aspect-oriented programming, in: AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development, ACM Press, New York, NY, USA, 2002, p. 141–147.

[9] M. Nishizawa, S. Chiba, M. Tatsubori, Remote pointcut: a language construct for distributed AOP, in: Proceedings of the 3rd international conference on Aspect-oriented software development, ACM, Lancaster, UK, 2004, pp. 7–15.

[10] G. Rozenberg (Ed.), Handbook of graph grammars and computing by graph transformation, Vol. 1-3, World Scientific Publishing Co., Inc., 1997.

[11] H. Schippers, D. Janssens, M. Haupt, R. Hirschfeld, Delegation-based semantics for modularizing crosscutting concerns, SIGPLAN Notices 43 (10) (2008) 525–542.

[12] M. S. Miller, E. D. Tribble, J. Shapiro, H. P. Laboratories, Concurrency among strangers: Programming in e as plan coordination, In Trustworthy Global Computing, International Symposium, TGC 2005 (2005) 195—229.

[13] J. H. C. Baker, C. Hewitt, The incremental garbage collection of processes, in: Proceedings of the 1977 symposium on Artificial intelligence and programming languages, ACM, 1977, pp. 55–59.

[14] D. Alhadidi, N. Belblidia, M. Debbabi, P. Bhattacharya, An AOP extended Lambda-Calculus, in: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society, 2007, pp. 183–194.

[15] S. D. Djoko, R. Douence, P. Fradet, D. L. Botlan, CASB: common aspect semantics base, Tech. Rep. AOSD-Europe Deliverable D41, AOSD-Europe-INRIA-7, INRIA, France (2006).

[16] B. D. Fraine, E. Ernst, M. Südholt, Essential AOP: the a calculus, in: ECOOP, 2010, pp. 101–125.

[17] R. Jagadeesan, A. Jeffrey, J. Riely, A calculus of untyped Aspect-Oriented programs., in: L. Cardelli (Ed.), ECOOP 2003 - Object-Oriented Programming, 17th European Conference, 2003, pp. 54–73.

[18] T. Staijen, A. Rensink, Graph-based specification and simulation of featherweight java with around advice, in: Proceedings of the 2009 workshop on Foundations of aspect-oriented languages, ACM, Charlottesville, Virginia, USA, 2009, pp. 25–30.

[19] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. Liao, C. Tseng, M. W. Hall, M. S. Lam, J. L. Hennessy, SUIF: an infrastructure for research on parallelizing and optimizing compilers, SIGPLAN Not. 29 (12) (1994) 31–37.

[20] G. Necula, S. McPeak, S. Rahul, W. Weimer, CIL: intermediate language and tools for analysis and transformation of c programs, in: Compiler Construction, 2002, pp. 209–265.

[21] F. Gadducci, Graph rewriting for the pi-calculus, Mathematical Structures in Computer Science 17 (3) (2007) 407–437.

[22] P. Baldan, H. Ehrig, J. Padberg, G. Rozenberg, Workshop on petri nets and graph transformations, in: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (Eds.), Graph Transformations, Vol. 4178 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 467–469, 10.1007/11841883_35.

# A Visual Analytics Toolset for Program Structure, Metrics, and Evolution Comprehension

Dennie Reniers[a], Lucian Voinea[a], Ozan Ersoy[b], Alexandru Telea[b,*]

[a] SolidSource BV, Eindhoven, the Netherlands
[b] Institute Johann Bernoulli, University of Groningen, the Netherlands

**Abstract**

Software visual analytics (SVA) tools combine static program analysis and repository mining with information visualization to support program comprehension. However, building efficient and effective SVA tools is highly challenging, as it involves software development in program analysis, graphics, information visualization, and interaction. We present a SVA toolset for software maintenance, and detail two of its components which target software structure, metrics and code duplication. We illustrate the toolkit's application with several use cases, discuss the design evolution of our toolset from a set of research prototypes to an integrated, scalable, and easy-to-use product, and argue how these can serve as guidelines for the development of future software visual analytics solutions.

*Keywords:* Software visualization, static analysis, visual tool design

## 1. Introduction

Software maintenance covers 80% of the cost of modern software systems, of which over 40% represent software understanding [1, 2]. Although many visual tools for software understanding exist, most know very limited acceptance in the IT industry. Key reasons for this are limited scalability of visualizations and/or size of datasets, long learning curves, and poor integration with established software analysis or development toolchains [3, 4, 5].

*Visual analytics* (VA) integrates graphics, visualization, interaction, data analysis, and data mining to support reasoning and sensemaking for complex problem solving in engineering, finances, security, and geosciences [6, 7]. These fields share many similarities with software maintenance in terms of *data* (large databases, highly structured text, and graphs), *tasks* (sensemaking by hypothesis creation, refinement, and validation), and *tools* (combined analysis and visualization). VA explicitly addresses tool scalability and integration, as opposed to pure data mining (whose main focus is scalability) or information visualization (InfoVis, mainly focusing on presentation). As such, VA is a promising model for building effective and efficient software visual analysis (SVA) tools. However, building VA solutions for software comprehension is particularly challenging, as developers have to master technologies as varied as static analysis, data mining, graphics, information visualization, and user interaction design.

In this paper, we present our experience in building SVA tools for software maintenance. We outline the evolution path from a set of research prototypes to a commercial toolset which is used by many end-users in the IT industry. Our toolset supports static analysis, quality metrics computation, clone detection, and research-grade InfoVis techniques such as table lenses, bundled graph layouts, cushion treemaps, and dense pixel charts. The toolset contains several end-user applications, of which we focus here on two: visual analysis of program structure and code duplication. These applications share elements at both design and implementation level, and can be used separately or combined to support tasks such as detecting correlations of structure, dependencies, and quality metrics; assessing system modularity; and planning refactoring.

*Corresponding author

*Email addresses:* dennie.reniers@solidsource.nl (Dennie Reniers), lucian.voinea@solidsource.nl (Lucian Voinea), o.ersoy@rug.nl (Ozan Ersoy), a.c.telea@rug.nl (Alexandru Telea)

*URL:* www.solidsource.nl (Dennie Reniers), www.solidsource.nl (Lucian Voinea), www.cs.rug.nl/~alext (Alexandru Telea)

The contributions of this paper are as follows:

- present the design decisions and evolution path of a SVA toolset for program comprehension from research prototypes into an actual product;
- present the lessons learned in developing our toolset in both research and industrial contexts, with a focus on efficiency, effectiveness, and acceptance;
- present supporting evidence for our design decisions based on actual usage in practice.

Section 2 introduces software visual analytics. Section 3 details the general architecture of our toolset. Section 4 details the toolset's components for data mining and visualization of software structure, metrics, and code duplicates. Section 5 illustrates the usage of our toolset in a real-world industrial software assessment case. Section 6 discusses the challenges of developing efficient, effective, and accepted SVA tools from our toolset's experience. Finally, section 7 concludes the paper.

## 2. Related Work

Software visual analytics can be roughly divided into data mining and visualization, as follows.

*Data mining* covers the extraction of raw data from source code, binaries, and source control management (SCM) systems such as CVS, Subversion, Git, CM/Synergy, or ClearCase. Raw data delivered by static syntax and semantic analysis is refined into structures such as call graphs, control flow graphs, program slices, code duplicates (clones), software quality metrics such as complexity, cohesion, and coupling, and change patterns. Static analyzers can be divided into *lightweight* ones, which use a subset of the target language grammar and semantics and typically trade fact completeness and accuracy for speed and memory; and *heavyweight* ones, which do full syntactic and semantic analysis at higher cost. Well-known static analyzers include LLVM, Cppx, Columbus, Eclipse CDT, and Elsa (for C/C++), Recoder (for Java), and ASF+SDF (a meta-framework usable with different language-specific front-ends) [8]. Metric tools include CodeCrawler [9], Understand, and Visual Studio Test Suite (VSTS). An overview of static analysis tools with a focus on C/C++ is given in [10]. Practical insights in software evolution and software quality metrics are given in [11, 12, 13].

*Software visualization* (SoftVis) uses information visualization (InfoVis) techniques to create interactive displays of software structure, behavior, and evolution. Recent trends in SoftVis include scalable InfoVis techniques such as treemaps, icicle plots, bundled graph layouts, table lenses, parallel coordinates, multidimensional scaling, and dense pixel charts to increase the amount of data shown to the user at a single time. An excellent overview of software visualization is given in [14]. Well-known software visualization systems include Rigi, VCG, aiSee, and sv3D (for structure and metrics); and CodeCity and SeeSoft (for software evolution).

Although tens of new software analysis and visualization tools emerge every year from research, as shown by the proceedings of *e.g.* ACM SOFTVIS, IEEE VISSOFT, MSR, ICPC, WCRE, and CSMR, building *useful* and *used* tools is difficult. Reasons cited for this include the small return-on-investment and recognition of tools in academia (as opposed to papers), high maintenance cost, and high ratio of infrastructure-to-novelty (truly usable tools need fine-tuned implementations, help modules, and platform-independence, while research prototypes can focus on novelty) [5, 15]. Combining analysis and visualization in the same tool makes development only more complex, so good design patterns and guidelines are essential.

Given all these, how to bridge the gap between SVA tool prototypes and actual efficient and effective products? And how to combine analysis and visualization software in maintainable tool designs?

## 3. Toolset Architecture

In the past decade, we have built over 20 SVA tools for software requirements, architecture, behavior, source code, structure, dependencies, and evolution. We used these tools in academic classes, research, and industry, in groups from a few to tens of users. Latest versions of our tools have formed the basis of SolidSource, a company specialized in software visual analytics [16]. Table 1 outlines our most important tools (for a full list and the actual tools, see [17]). We next present the latest version of our toolset, discuss the design decisions and lessons learned during the several years of its evolution, and illustrate its working with two of its most recent tools.

| Tool | Targeted data types | Visual techniques | Analysis techniques |
|---|---|---|---|
| SoftVision (2002) [18] | software architecture | node-link layouts (2D and 3D) | none (visualization tool only) |
| CSV (2004) [19] | source code syntax and metrics | pixel text, cushions | C++ static analysis (`gcc` based parser) |
| CVSscan [20] (2005) | file-level evolution | dense pixel charts annotated text | CVS data mining (authors and line-level changes) |
| CVSgrab (2006) [21] | project-level evolution | dense pixel charts, cushions | CVS/SVN data mining (project-level changes) |
| MetricView (2006) [22] | UML diagrams and quality metrics | node-link layouts (2D), table lenses | C++ lightweight static analysis (class diagram extraction) |
| MemoView (2007) [23] | software dynamic logs (memory allocations) | table lenses, timelines, cushions | C runtime instrumentation (`libc` malloc/free interception) |
| SolidBA (2007) [24] | build dependencies and build cost metrics | table lenses, node-link layouts (2D) | C/C++ build dependency mining and automated refactoring |
| SolidFX (2008) [25] | reverse engineering | pixel text, table lenses, annotated text | C/C++ heavyweight error-tolerant static static analysis |
| SolidSTA (2008) [16] | file and project-level evolution | dense pixel charts, cushions, timelines | CVS/SVN/Git data mining and source code metrics |
| **SolidSX** (2009) [16] | structure, associations, metrics | HEB layouts, treemaps, table lenses, cushions | .NET, C++, Java lightweight static analysis |
| **SolidSDD** (2010) [16] | code duplicates, code structure, metrics | HEB layouts, treemaps, table lenses, pixel text | C, C++, Java, C# parameterizable syntax-aware clone detection |

Table 1: Software visual analytics tools - evolution history. Tools discussed in this paper are in bold (Sec. 4)

Our toolset uses a simple dataflow architecture (Fig. 1). Raw input data comes in two forms: non-versioned source code or binaries, and versioned files stored software repositories. From raw data, we extract several *facts*: syntactic and semantic structure, static dependency graphs, and source code duplication. Relational data is stored into a SQLite database whose entries point to flat text files (for actual source code content) and binary files (for complete syntax trees, see 4.1.1). Fact extraction is implemented by specific tools: parsers and semantic analyzers for source code, binary analyzers for binary code, and clone detectors for code duplication (see Sec. 4).

Besides facts, our database stores two other key elements: selections and metrics. *Selections* are sets of facts (or other selections) and support the iterative data refinement in the so-called visual sensemaking cycle of VA [6, 7]. They are created either by tools, *e.g.* extraction of class hierarchies or call graphs from annotated syntax graphs (ASGs), or by users during interactive data analysis. Selections have unique names by which they are referred by their clients (tools) and also persistently saved. *Metrics* are numerical, ordinal, or categorical attributes which can be associated to facts or selections, and are computed either by tools (*e.g.* complexity, fan-in, fan-out, cohesion, and coupling) or added as annotations by users. Each selection is stored as a separate SQL table, whose rows are its facts, and columns are the fact attributes (unique ID, type, location in code or binary, and metrics). This model is simple and scales well to fact databases of hundreds of thousands of facts with tens of metrics. Trees and graphs can also be stored as SQL tables in terms of adjacency matrices. Simple queries and filters can directly be implemented in SQL. More complex queries have the same interface as the other components: they read one or more selections and create an output selection.

Selections are the glue that allows composing multiple analysis and visualization tools. All analysis and visualization components in our toolkit are weakly typed: They all read selections, and optionally create selections (see Fig. 1). In this way, existing or new tools can be flexibly composed, either statically or at run-time, with virtually no configuration costs. To ensure consistency, each component decides internally whether (and how) it can execute its function on a given input selection.

*Visualizations* display selections and allow users to interactively navigate, pick elements, and customize the visual aspect. Since they only receive selections as inputs, they 'pull' their required data on demand as needed, *e.g.* a source code viewer opens the files referred by its input selection to get the text to render. Hence, data can be efficiently handled by reference (selections referred to by name), but tools can also decide to cache data internally to minimize traffic with the fact database, if so desired. However, this decision is completely transparent at the toolkit level.

After lengthy experimentation with many types of visualizations, we limited ourselves to a small subset hereof, as follows. *Table lenses* show large tables by zooming out the table layout and displaying cells as pixel bars scaled and colored by data

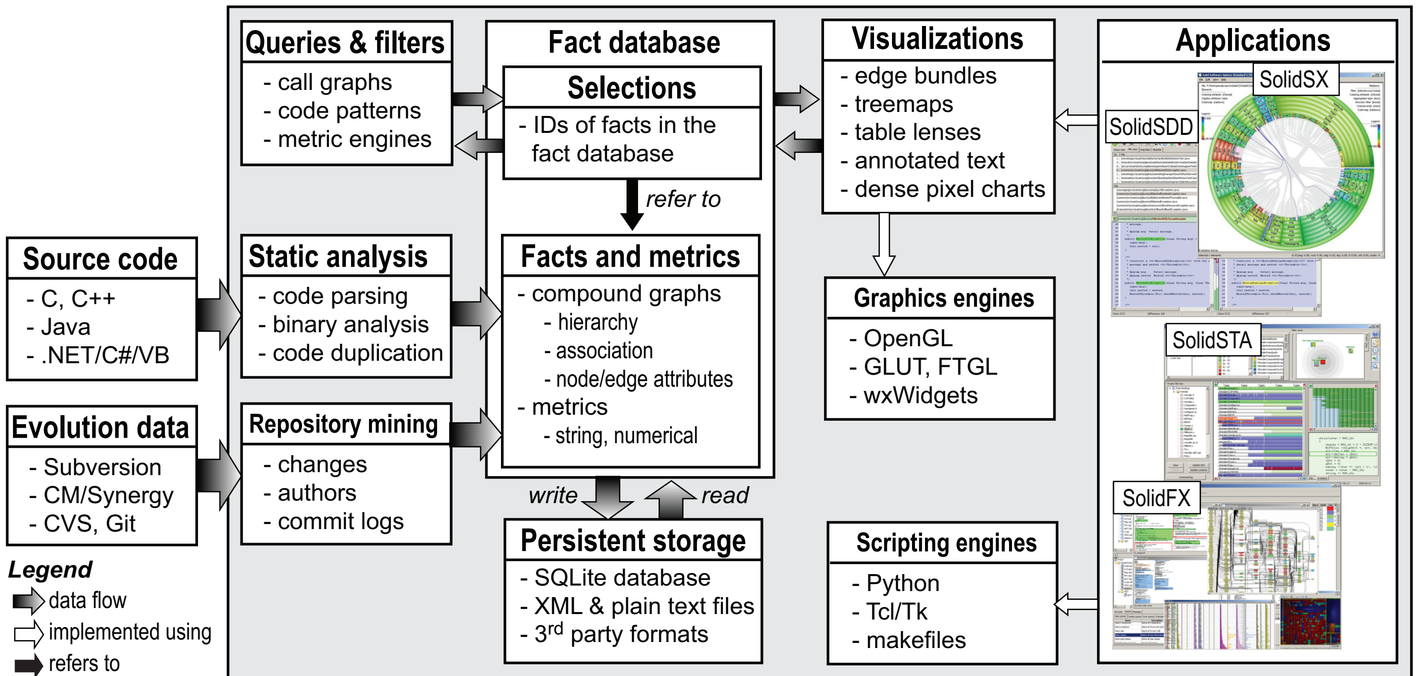



Figure 1: Toolset architecture (see Section 3)

values [26]. Subpixel sampling techniques enhance this idea allowing to visualize tables up to hundreds of thousands of rows on a single screen [27]. *Hierarchically bundled edges* (HEBs) compactly show compound (structure and association) software graphs, *e.g.* containinment and call relations, by bundling association edges with structure data [28]. *Squarified cushion treemaps* show software structure and metrics scalably for up to tens of thousands of elements on a single screen [29].

A single treemap, HEB, or table lens can show the correlation of just a few metrics and/or relations. To augment this, we use the well-known *multiple correlated views* InfoVis concept. Each view can have its own user-specified input selection and visual look-and-feel. Mouse-based picking is synchronized between all views: selecting items in one view updates a 'user selection' which is monitored by other views, using an Observer pattern. Besides the above, our toolset also provides classical visualizations: metric-annotated source code text, tree browsers, customizable color maps, legends, annotations, timelines, details-on-demand at the mouse cursor, and text and metric-based search facilities.

## 4. Toolset Components

Our toolset consists of several end-user applications which share analysis and visualization components. We next describe two of these applications: the SolidSX structure analyzer (Sec. 4.1) and the SolidSDD duplication detector (Sec. 4.1). The fully operational tools, including Windows-based installers, manuals, videos, and sample datasets are freely available for researchers from [16].

### *4.1. SolidSX: Structural Analysis*

The SolidSX (Software eXplorer) supports the task of analyzing software structure, dependencies, and metrics. Several built-in parsers are provided: Recoder for Java source and bytecode files [30], Reflector for .NET/C# assemblies [31], and Microsoft's free parser for Visual C++ *.bsc* symbol files. Built-in filters refine parsed data into a compound attributed graph consisting of one or several hierarchies, *e/g.* folder-file-class-method containment and namespace-class-method containment; dependencies, *e.g.* calls, symbol usage, inheritance, and package or header inclusion; and basic metrics, *e.g.* code and comment size, cyclomatic complexity, fan-in, fan-out, and symbol source code location.

#### *4.1.1. Static Analysis: Ease of Use*

Setting up static code analysis is notoriously complex, error-prone, and time consuming. For .NET/VB/C#, Java, and Visual C++, we succeeded in *completely* automating this process. The user is only required to input a root directory for code and, for Java, optional class paths. C/C++ static analysis beyond Visual Studio proves highly challenging. SolidSX can read static information created by our separate SolidFX C/C++ static analyzer [25]. Although SolidFX is scalable to millions of lines of code, covers several dialects beyond Visual C++ (gcc, C89/99, ANSI C++), robustly handles incorrect and incomplete code, integrates a preprocessor, provides a Visual C++ project file parser, and uses so-called compiler wrapping to emulate the `gcc` and Visual C++ compilers [32], users still typically need to manually supply many per-project defines and include paths. Moreover, compiler wrapping requires the availability of a working build system on the target machine. The same is true for other heavyweight parsers such as Columbus [32] or Clang [33].

A second design choice regards performance. For Java, Recoder is close to ideal, as it delivers heavyweight information with 100 KLOC/second parse speed on a typical PC computer. Strictly speaking, visual structure-and-metric analysis only needs lightweight analysis (which is fast) as the information graininess typically does not go below function level. For .NET, the Reflector lightweight analyzer is fast, robust, and simple to use. The same holds for Microsoft's *.bsc* symbol file parser.

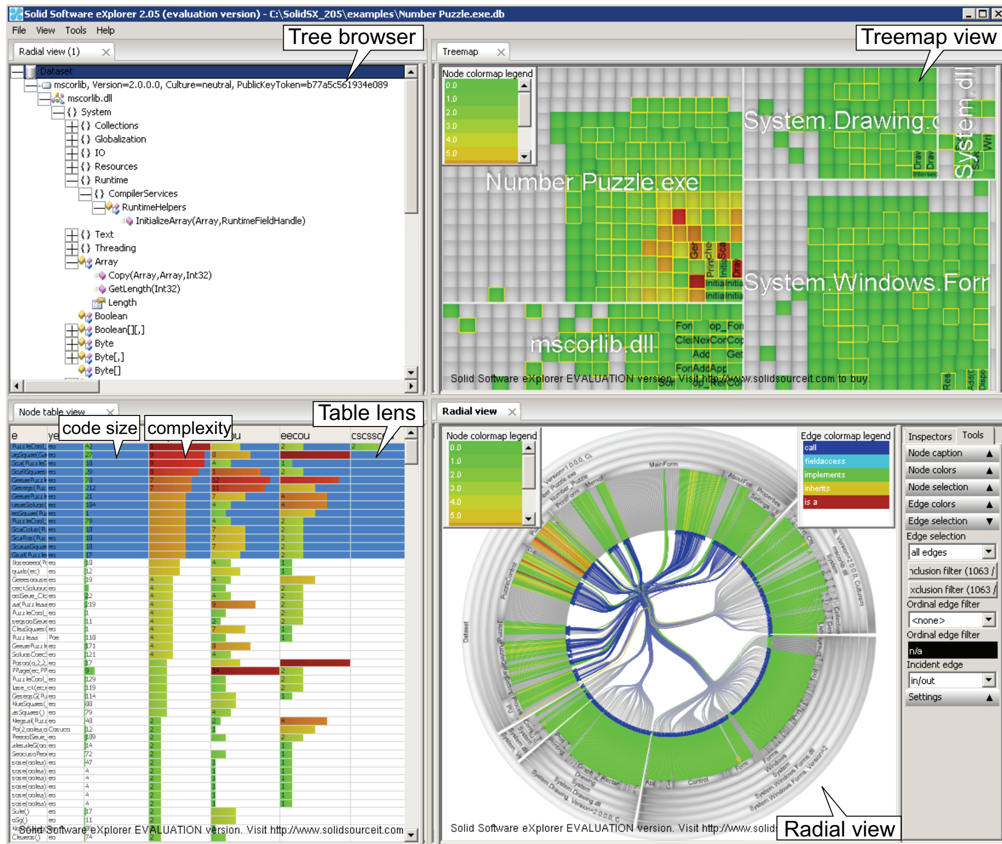
For C/C++ beyond Visual Studio, a lightweight, robust, easy-to-use analyzer is still not available. After experimenting with many such tools, *e.g.* CPPX [34], gccxml, and MC++, we found that these often deliver massively incorrect information, mainly due to simplified preprocessing and name lookup implementations. The built-in C/C++ parsers of Eclipse CDT, KDevelop, QtCreator, and Cscope [35] are somehow better in correctness, and can also work incrementally upon code changes. However, these parsers depend in complex ways on their host IDEs and do not have well-documented APIs. Extended discussions with Roberto Raggi, the creator of the KDevelop and QtCreator parsers, confirmed this difficulty.

#### *4.1.2. Structure Visualization*

SolidSX offers four views (Fig. 2 top): classical tree browsers, table lenses of node and edge metrics, treemaps, and a HEB layout for compound graphs [28]. All visualizations have carefully designed *presets* which allow one to use them with no additional customization. They all depict selections from the fact database created by the code analysis step. Users can also create selections in any view by either direct interaction or custom queries. These two mechanisms realize the linked view concept, which enables users to easily create complex analyses of correlations of structure, dependencies, and metrics along different viewpoints. Figure 2 top) illustrates this on a C# system of around 45000 LOC (provided with the tool distribution). The HEB view shows function calls over system structure: caller edge ends are blue, callee edge ends are gray. Node colors show McCabe's code complexity metric on a green-to-red colormap, thereby enabling complexity correlation with the system structure. We see that the most complex functions (warm colors) are in the module and classes located top-left in the HEB layout. The table lens view shows several function-level code metrics, and is sorted on decreasing complexity. This allows one to see how different metrics correlate with each other. Alternatively, one can select *e.g.* the most complex or largest functions and see them highlighted in the other views. The treemap view shows a flattened system hierarchy (modules and functions only), with functions ordered top-down and left-to-right in their parent modules on code size, and colored on complexity. The visible 'hot spot' indicates that complexity correlates well with size. Constructing the *entire* scenario, including the static analysis, takes about 2 minutes and under 20 mouse clicks.

#### *4.1.3. Toolchain Integration*

Similarly to Koschke [5], we discovered that *integration* in accepted toolchains is a key acceptance factor. To ease this process, we added a *listener* mechanism to SolidSX. The tool listens for command events sent asynchronously on a specified TCP/IP port, *e.g.* load a dataset, zoom on some data subset, change view parameters, and also sends user interaction command events to a second port (*e.g.* user has selected some data). This allows integrating SolidSX in any third-party tool or toolchain by building thin wrappers which read, write, and process desired events. No access to the tool's source code is required. For example, we integrated SolidSX in the Visual Studio IDE by writing a thin plug-in of around 200 LOC which translates between the IDE and SolidSX events (Fig. 2 bottom). Selecting and browsing code in the two tools is now in sync. The open SQLite database format further simplifies integration at data level. Integrating SolidSX in Eclipse, KDevelop, and QtCreator is under way and is relatively easy, once we finalize the data importers from these IDEs' proprietary fact databases into our SQL database.



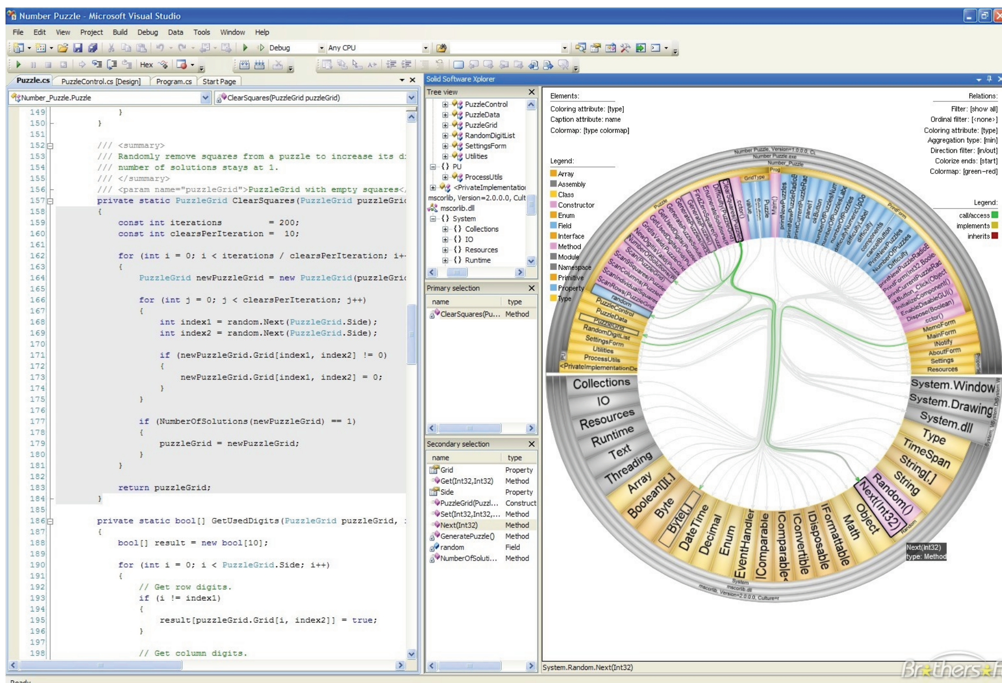




Figure 2: Top: SolidSX views (tree browser, treemap, table lens, radial HEB); Bottom: Visual Studio integration of SolidSX using a socket-baset mechanism (see Section 4.1.3)

*4.2. SolidSDD: Clone Detection*

Code duplication detection, or clone detection, is an important step in maintenance tasks such as refactoring, redocumentation, architecture extraction, and test case management. Although hundreds of papers on this topic exist, only few clone detection *tools* offer the clone information in effective ways for assisting refactoring activities.

We addressed the above by developing SolidSDD (Software Duplication Detector). SolidSDD implements an extended version of the CCfinder tool [36] which combines lightweight syntax analysis and token-based detection. The detection is user-configurable by clone length (in statements), identifier renaming (allowed or not), size of gaps (inserted or deleted code fragments in a clone), and whitespace and comment filtering. However, the main novelty in SolidSDD is in how clones are visualized.

Given a source code tree (C, C++, Java, or C#), SolidSDD constructs a SQL database containing a duplication graph, in which nodes are cloned code fragments (in the same or different files) and edges indicate clone relations. Hierarchy information is added to this graph either automatically (from the code directory structure) or from a user-supplied dataset (*e.g.* from static analysis). The result is a compound graph. Nodes and edges can have metrics, *e.g.* percentage of cloned code, number of distinct clones, and whether a clone includes identifier renaming or not. Metrics are automatically aggregated bottom-up using the hierarchy information.

Figure 3 shows SolidSDD in use to find code clones in the well-known Visualization Toolkit (VTK) library [37]. On VTK version 5.4 (2420 C++ files, 668 C files, and 2660 headers, 2.1 MLOC in total), SolidSDD found 946 clones in 280 seconds on a 3 GHz PC with 4 GB RAM. For replication ease, we simply used the default clone detection settings of the tool. Figure 3 a shows the code clones atop of the system structure with SolidSX. Hierarchy shows the VTK directory structure, with files as leaves. Edges show aggregated clone relations between files: file $a$ is connected to file $b$ when $a$ and $b$ share at least one clone. Node colors show the total percentage of cloned code in the respective subsystem with a green-to-red (0..100%) colormap. Edge colors show percentage of cloned code in the clone relations aggregated in an edge. Figure 3 a already shows that VTK has quite many intra-system clones (short edges that start and end atop of the same rectangle) but also several inter-system clones (longer edges that connect rectangles located in different parts of the radial plot). Inter-system clones are arguably less desirable.

Three subsystems have high clone percentages: *examples* ($S_1$), *bin* ($S_2$) and *Filtering* ($S_3$). Browsing the actual files, we saw that almost all clones in *examples* and *bin* are in sample code files and test drivers which contain large amounts of copy-and-paste. Clones in *Filtering*, a core subsystem of VTK, are arguably more worth removing. In Fig. 3 b, we use SolidSX's zoom and filter functions to focus on this subsystem and select one of its high-clone-percentage files $f$ (marked in black) which has over 50% cloned code. When we select $f$, only its clone relations are shown. We call the set of files $f_c$ with which $f$ shares clones the *clone partners* of $f$. We see that all clone partners of $f$ are located in the same *Filtering* subsystem, except one ($g$) which is in the *Rendering* subsystem.

Figure 3 c shows additional functions offered by SolidSDD. The top light-blue table shows all system files with several metrics: percentage of cloned code, number of clones, and presence of identifier renaming in clones. SolidSDD's views are linked with SolidSX's views via the socket mechanism outlined in Sec. 4.1.3, so the selected file $f$ in Fig. 3 c is also highlighted in this table in red. The table below shows the clone partner files $f_c$ of $f$. In this table, we can identify the file $g$ which shares clones with $f$ but is in another subsystem. We also see here that $g$ contains about 50% of code cloned from $f$. We select $g$ and use the two text views (at the bottom of Fig. 3 b) to examine in detail all clones between $f$ and $g$. The left view shows the first selected file ($f$) and the right view the selected clone partner ($g$). Scrolling of these views is automatically synchronized so one can easily follow corresponding code fragments. Text is color-coded as follows: non-cloned code (white), code from $f$ which is cloned in $g$ (light blue), renamed identifier pairs (green in left view, yellow in right view), and code from $f$ which is cloned but whose clones are located in some other file than $g$ (light brown). The last color allows us to navigate from $f$ to other clone partners: Clicking on the light brown code in the left text view ($f$) in Fig. 3 c replaces the file $g$ in the right view by that clone partner of $f$ (let us call it $h$) with which $f$ shares the brown code, and also selects $h$ in the clone partner list view. Fig. 3 d shows this perspective. We now notice that the code in $f$ which is part of the clone $f - g$ (light blue in Fig. 3 c) is *included* in the clone $f - h$ (light blue Fig. 3 c).

The SolidSDD-SolidSX combination offers many other perspectives and analyses, as detailed by its manual. Fig. 3 e shows a final example. We use SolidSX's table lens to sort all files by percentage of cloned code and zoom out to see a distribution of this metric (sixth column from right). We see that around 30% of all files contain some amount of cloning. Interactively selecting the top 20% files in the table lens highlights all files having >80% cloned code in the radial system view in black - we

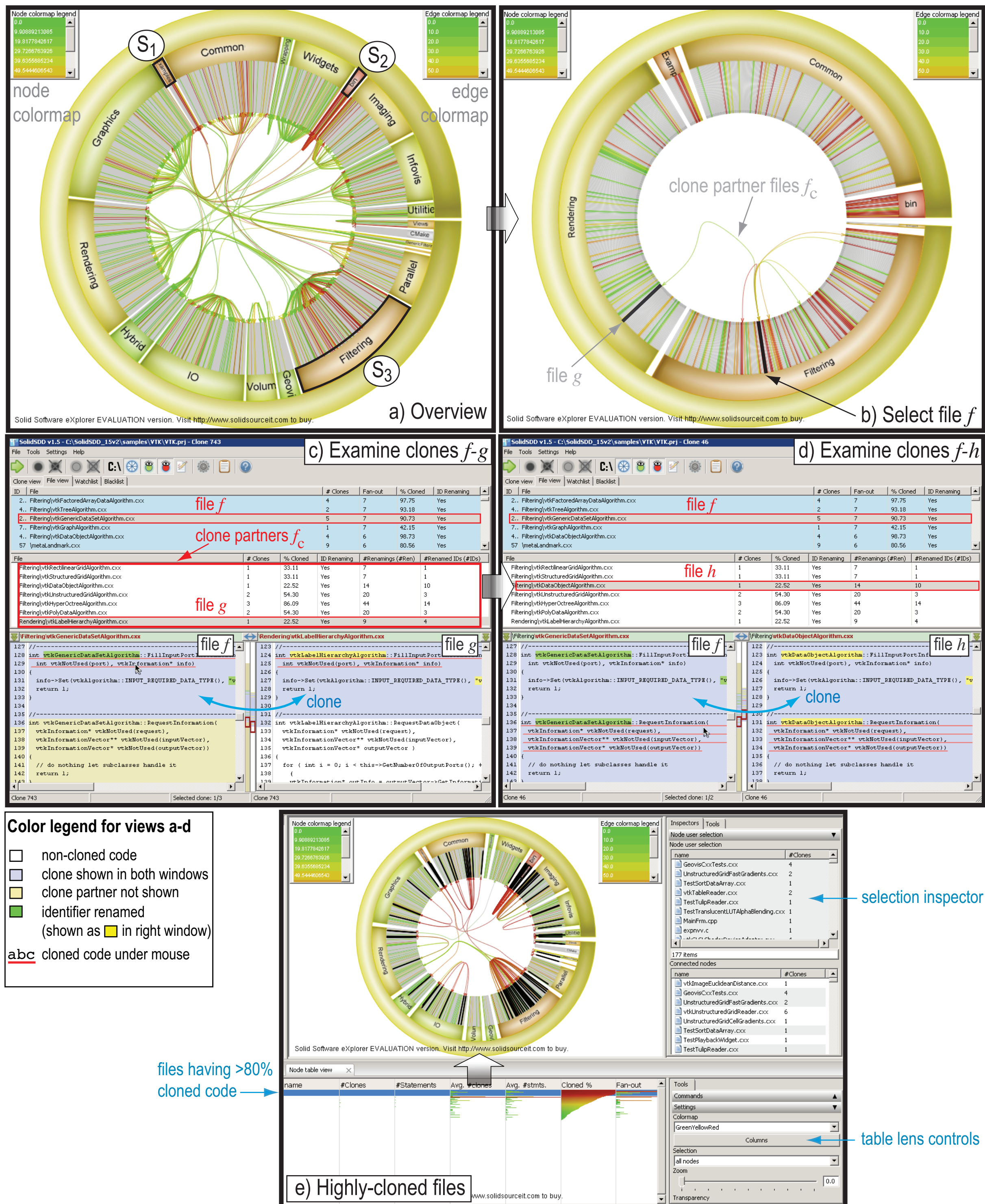



Figure 3: SolidSDD clone detector visual analysis of the VTK code base (see Sec. 4.2)

see that every single subsystem in VTK contains such files. Details on the selected files are shown in the selection inspector at the right. This type of visualization supports refactoring planning by giving insight on where recoding activities would need to take place in order to *e.g.* remove the largest clones.

## 5. Toolset Assessment in Practice

We have used our SVA toolset in many applications, both in research [19, 38] and the industry [24, 39, 25]. We next briefly describe one application which outlines the added value of strong interplay between the two tools discussed so far. To further outline the extensibility of our toolset, this application briefly introduces one additional tool: SolidSTA, a visual analytics application for software evolution data mining. SolidSTA uses the same dataflow architecture (Sec. 3) and followed the same development path from a research prototype [21] to a scalable, easy-to-use tool as SolidSX and SolidSDD. A detailed description of SolidSTA is provided in [21]. The tool is freely available for researchers from [16].

A major automotive company developed an embedded software stack of 3.5 million lines of C code in 15 releases over 6 years with three developer teams in Western Europe, Eastern Europe, and Asia. Towards the end, it was seen that the project could not be finished on schedule and that new features were hard to introduce. The management was not sure what went wrong. The main questions were: was the failure caused by bad architecture, coding, or management; and how to follow up - start from scratch or redesign the existing code. An external consultant team performed a post-mortem analysis using our toolset (SolidSDD, SolidSTA, SolidSX). The team had only *one week* to deliver its findings and only the code repository as information source. For full details, see [40].

The approach involved the classical VA steps: data acquisition, hypothesis creation, refinement, (in)validation, and result aggregation and presentation (Fig. 4). First, we mined change requests (CRs), commit authors, static quality metrics, and call and dependency graphs from the CM/Synergy repository into our toolset's SQL fact database using SolidSTA (1). Next, we examined the distribution of CRs over project structure. Several folders with many open CRs emerged (red treemap cells in Fig. 4 (2)). These correlate quite well with the team structure: the 'red' team owns most CRs (3). To further see if this is a problem, we looked at the CR distribution over files over time. In Fig. 4 (4), files are shown as gray lines vertically stacked on age (oldest at bottom), and CRs are red dots (the same layout is used *e.g.* in [21]). The gray area's shape shows almost no project size increase in the second project half, but many red dots over *all* files in this phase. These are CRs involving old files that were never closed. When seeing these images, the managers instantly recalled that the 'red' team (located in Asia) had lasting communication problems with the European teams, and acknowledged that it was a mistake to assign so many CRs to this team.

We next analyzed the evolution of various quality metrics: fan-in, fan-out, number of functions and function calls, and average and total McCabe complexity. Static analysis is done using our heavyweight analyzer SolidFX [25]. Since this is a one-time analysis rather than an incremental change-and-reanalyze path, speed and configuration cost are not essential, so most of the analyzers listed in Sec. 4.1.1 could be used equally well. The graphs in (5) show that these metrics have a slow or no increase in the second project half. Hence, the missed deadlines were not caused by code size or complexity explosion. Yet, the average complexity per function is high, which implies difficult testing. This was further confirmed by the project leader.

Finally, to identify possible refactoring problems, we analyzed the project structure using SolidSX. Fig. 4 (6) shows disallowed dependencies, *i.e.* modules that interact bypassing interfaces. Fig. 4 (7) shows modules related by mutual calls, which violate the product's desired strict architectural layering. These two views suggest difficult step-by-step refactoring and also difficult unit testing. Again, these findings were confirmed by the project leaders.

## 6. Discussion

Based on our SVA tool building experience, we next try to answer several questions of interest for our audience[1]. To clarify the standpoints taken, we first introduce the concept of a tool *value model*, along the lines of the well-known lean development cost model [41]: We state that a SVA tool is useful if it delivers high *value* with minimal *waste* to its stakeholder user. Different users (or stakeholders) will thus assess the same tool differently as they have different value and waste models [42]. Hence, rhe answers to our questions of interest are strongly dependent on the stakeholder perspective chosen, as follows.

[1] as taken from the WASDeTT 2010 call for papers (www.info.fundp.ac.be/wasdett2010)

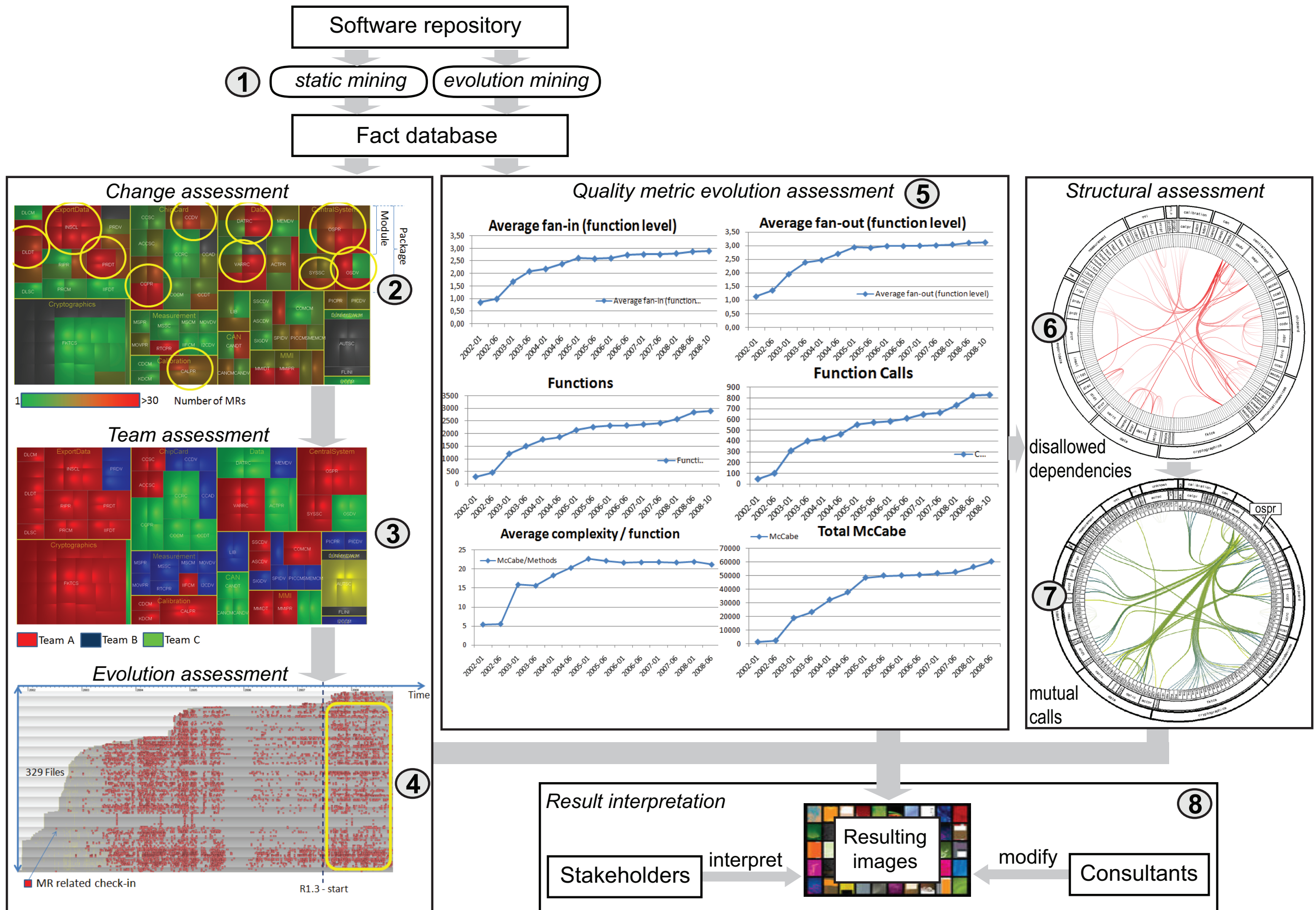



Figure 4: Data collection, hypothesis forming, and result interpretation for product and process assessment (Sec. 5). Arrows and numbers indicate the order of the performed steps

### *6.1. Should academic tools be of commercial quality?*

We see two main situations here. If tools are used by researchers *purely* to test new algorithms or ideas, *e.g.* a new visualization layout atop of the HEB implemented in SolidSX [38], then large investments in tool infrastructure (Sec. 2) is seen as waste. If tools are needed in case studies with external users or, even stronger, in real-life projects, then their usability is key to acceptance and success [5, 43, 39]. For example, valuable insight obtained from user-testing our toolset on large software projects with more than 70 students, 15 researchers, and 30 IT developers could never have been reached if our tools had not been mature enough for users to accept working with them in the first place. Hence, we believe that academic tools intended for other users than their immediate researcher creators should not compromise on *critical* usability (*e.g.* interactivity, scalability, robustness). However, effort required for *adoptability* (*e.g.* manuals, installers, how-to's, support of many input/output formats, rich GUIs), which is critical only in later deployment phases, can be limited.

### *6.2. How to integrate and combine independently developed tools?*

This is an extremely challenging question as both the integration degree required and the tool heterogeneity vary widely in practice. For SVA tools, we have noticed the following practical patterns to provide good returns on investment, in increasing order of difficulty:

- *dataflow:* Tools communicate by reading and writing several data files in standardized formats, *e.g.* SQL for tables, GXL and XML (for attributed graphs) [44], and FAMIX and XMI (for design and architecture models) [45]. This easily allows creating dataflow-like tool pipelines, like the excellent Cpp2Xmi UML diagram extractor involving Columbus and Graphviz [46].

- *shared databases:* Tools communicate by reading and writing a single shared fact database which stores code, metrics, and relations, typically as a combination of text files (for code), XML (for lightweight structured data), and proprietary binary formats (for large datasets such as ASGs or execution traces). This is essentially the model used by Eclipse's CDT, Visual Studio's Intellisense, and SolidSX. As opposed to dataflows, shared databases support the much finer-grained data access required by interactive visualization tools *e.g.* for real-time browsing of dependencies (SolidSX) or symbol queries (Eclipse, Visual Studio, SolidFX). However, adding third-party tools to such a database requires writing potentially complex data converters.

- *common API:* Tools communicate by using a centralized API *e.g.* for accessing shared databases but also executing operations on-demand. This allows functionality reuse and customization at a finer-grained level of 'components' rather than monolithic tools. Although a common API does not necessarily enforce a shared tool implementation (code base), the former typically implies the latter in practice. API examples for SVA tools are the Eclipse, Visual Studio, and CodeCrawler [9] SDKs and, at a lower level, the Prefuse and Mondrian toolkits for constructing InfoVis applications [47, 48]. Our own toolset also offers a common C++ API for the table lens, treemap, and HEB visualizations which allowed us to easily embed these in a wide range of applications, see *e.g.* [27, 39, 40]. Common APIs are more flexible than shared databases, as they allow control *and* data flow composition. However, they are much harder to use in practice, as they impose coding restrictions and high learning costs (for tool builders) and costly maintenance (for API providers).

In practice, most SVA toolsets use the dataflow or shared database model which nicely balances costs with benefits. This is the case of our toolset (see Sec. 3) and also the conceptually similar SQuAVisiT toolset [49]. The main difference between SQuAVisiT and our toolset is the integration level: in our case, analysis and visualization are tighter integrated, *e.g.* the built-in static analysis in SolidSX, clone detection in SolidSDD, SolidSX integration in Visual Studio, and clone visualization (SolidSX used by SolidSDD), realized by a single fact database, the event-based mechanism described in Sec. 4.1.3, and the shared C++ visualization components API.

*6.3. What are the lessons learned and pitfalls in building tools?*

SVA tool building is mainly a design activity. Probably the most important element for success is striving to create visual and interaction models which optimally fit the 'mental map' of the targeted users. Within space limitations, we outline the following points:

- *2D vs 3D:* software engineers are used to 2D visualizations, so they will accept these much easier than 3D ones [50]. We found no single case when a 3D visualization was better accepted than a 2D one in our work. As such, we abandoned earlier work in 3D visualizations [18] and focused on 2D visualizations only.

- *interaction:* too much interaction and user interface options confuse even the most patient users. A good solution is to offer problem-specific minimalist interaction paths or wizards, and hide the complex options under an 'advanced' tab. This design is visible in the user interfaces of both SolidSX and SolidSDD.

- *scalable integration* of analysis with visualization is absolutely crucial for the acceptance of the latter [5, 42]. However, providing this is very costly. We estimate that over 50% of the entire code base of our toolset (over 700 KLOC) is dedicated to efficient data representation for smooth integration. For datasets up to a few hundred thousand items, a SQLite fact database performs very well (Sec. 3). However, heavyweight parsing, such as performed by SolidFX, creates much larger datasets (full ASGs of tens of millions of elements, roughly 10..15 elements per line of code). To efficiently store and query this, we opted to store such data in a custom binary format which minimizes space and maximizes search speed [25]. The same is true for plain source code, which is best stored as separate text files. The SQL database is still used as a 'master' component which points to such special storage schemes for particular datasets. Finally, we use XML mainly as a data interchange format for lightweight datasets, *e.g.* SolidSX accepts all its input either as SQL or XML.

However, our experience is that XML does not lend itself well for storing arbitrary relational data for large datasets and efficiently implementing complex queries.

*6.4. What are effective techniques to improve the quality of academic tools?*

For VA tools, quality strongly depends on usability. Research tools aimed at quickly testing new visual algorithms should maximize API simplicity; here, Prefuse and Mondrian are good examples. Tools aimed at real-world software engineering problems should maximize end-user effectiveness. For SVA tools, this is further reflected into uncluttered, scalable, and responsive displays, and tight integration for short analysis-visualization sensemaking loops. Recent InfoVis advances have significantly improved the first points. However, integration remains hard, especially given that the academic 'value model' give tool-related studies relatively lesser credit than technical papers.

*6.5. Are there any useful tool building patterns for software engineering tools?*

For SVA tools, we see the following elements as present in most such tools we are aware of:

- *architecture:* the dataflow and shared database models are probably the widest used composition patterns.
- *visualizations:* 2D visualizations using dense pixel layouts like the table lens, HEBs, treemaps, and annotated text offer high scalability and ease of use, so are suitable for large datasets created from static analysis; node-link layouts generate too much clutter for graphs over about 1000 nodes and/or edges, but are better when position and shape encode specific meaning, like for (UML) diagrams (see further 6.6). Shaded cushions, originally promoted by treemaps [29], are a simple to implement, fast, visually scalable, and effective instrument of conveying structure atop of complex layouts.
- *integration:* tightly integrating independently developed analysis and visualization tools is still an open challenge. Although the socket-based mechanism outlined in Sec. 4.1.3 has its limitations, *e.g.* it cannot communicate shared state, it strikes a good balance between simplicity and keeping the software stacks of the several tools to integrate independent.
- *heavyweight vs lightweight analysis:* Lightweight static analyzers are considerably simpler to implement, deploy, and use, deliver faster performance, and may produce sufficient information for visualization (see Sec. 4.1.1). However, once visualization requirements increase, as it typically happens with a successful tool, so do the requirements on its input analyzer. Extending static analyzers is, however, not a simple process, and typically requires switching to a completely new analyzer tool. As such, keeping the fact database model simple and weakly typed offers more flexibility in switching analysis (and visualization) tool combinations.

*6.6. How to compare or benchmark such tools?*

Benchmarking SVA tools can be done by lab, class, or field user studies, or using the tool in actual IT projects, either by comparing several tools against each other [40] or by comparing a tool with a predefined set of desirable requirements [15, 43]. Measuring *technical* aspects *e.g.* speed, visual scalability, or analysis accuracy can be done using de facto standard datasets in the SoftVis community, *e.g.* the Mozilla Firefox, Azureus, or JHotDraw source code bases, which have been used in the ACM SoftVis, IEEE Vissoft, and IEEE MSR conference 'challenges'. Measuring a SVA tool's end-to-end *usefulness* is harder as it depends on task and context specific details; 'insight' and 'comprehension' are hard to quantify. Still, side-by-side tool comparison can be used. For example, Figure 5 shows four SVA tools (Ispace, CodePro Analytix, SonarJ, and SolidSX) whose effectiveness in supporting an industrial corrective maintenance task we compared [51, 42]. This, and similar, studies confirmed our supposition that node-link layouts are effective in program comprehension only for relatively small graphs (up to a few hundred nodes), which is the main reason why our current toolset focuses on the more scalable HEB layout. Other useful instruments in gathering qualitative feedback are 'piggybacking' the tool atop of an accepted toolchain (*e.g.* Eclipse or Visual Studio) and using established community blogs for getting user sentiment and success (or failure) stories. From our experience, we noticed that this technique works for both academic and commercial tools.

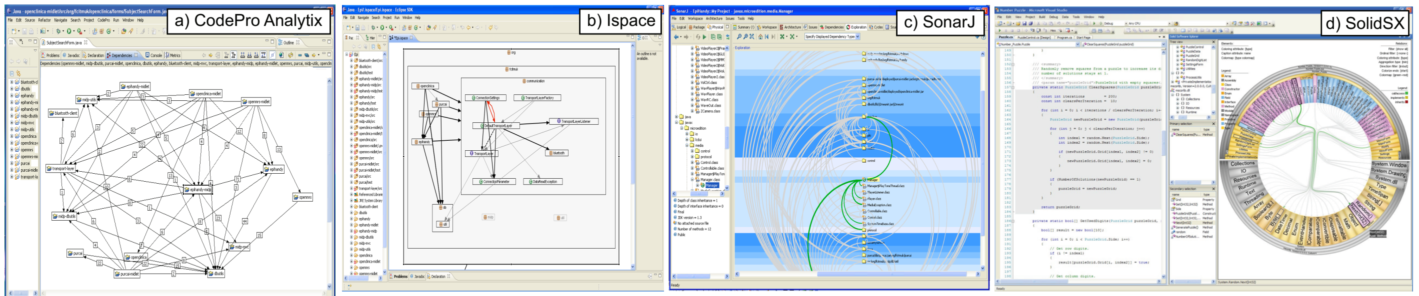



Figure 5: Four structure-and-dependency SVA tools compared for solving a corrective maintenance problem

### 6.7. *What particular languages and paradigms are suited to build tools?*

For SVA tools, our experience strongly converges to a minimal set of technologies, as follows:

- *graphics*: OpenGL, possibly augmented with simple pixel shader extensions, is by far the best solution in terms of portability, ease of coding, deployment, and performance; the same experience is shared by other researchers, *e.g.* Holten [28].
- *core:* scalability to millions of data items, relations, and attributes can only be achieved in programming languages like C, C++, or Delphi. Over 80% of all our analysis and application code is C++; Although Java is a good candidate, its performance and memory footprint are, from our experience, still not on par with compiled languages.
- *scripting:* Flexible configuration can be achieved in lightweight interpreted languages. The best candidate we found in terms of robustness, speed, portability, and ease of use was Python. Tcl/Tk (which we used earlier in [18] or Smalltalk (used by [9]) are also possible, but in our view require more effort for learning, deploying, and optimization.

## 7. Conclusions

In this paper, we have presented our experience in developing software visual analysis (SVA) tools, starting from research prototypes and ending with a commercial toolset. During this evolution and iterative design process, our toolset has converged from a wide variety of techniques to a relatively small set of proven concepts: the usage of a single shared weakly-typed fact database, implemented in SQL, which allows tool composition by means of shared fact selections; the usage of a small number of scalable InfoVis techniques such as table lenses, hierarchically bundled edge layouts, annotated text, timelines, and dense pixel charts; control flow composition by means of lightweight socket-based adapters as multiple linked-views in one or several independently developed tools; tool customization by means of Python scripts; and efficient core tool implementation using C/C++ and OpenGL.

We illustrated our toolset by means of two of its most recent applications: SolidSX for visualization of program structure, dependencies, and metrics, and SolidSDD for extraction and visualization of code clones. We outlined the added value of combining several tools in typical visual analysis scenarios by means of simple examples and a more complex industrial post-mortem software assessment case. Finally, from the experience gained in this development process, we addressed several questions relevant to the wider audience of academic tool builders.

Ongoing work targets the extension of our toolset at several levels: New static analyzers to support `gcc` binary analysis and lightweight zero-configuration C/C++ parsing; dynamic analysis for code coverage and execution metrics; and integration with the Eclipse IDE. Finally, we consider extending our visualizations with new metaphors which allow an easier navigation from source code to structural level by combining HEB layouts and annotated code text in a single scalable view.

## References


[1] T. A. Standish, An Essay on Software Reuse, IEEE TSE 10 (5) (1984) 494–497.
[2] T. Corbi, Program Understanding: Challenge for the 1990s, IBM Systems Journal 28 (2) (1999) 294–306.
[3] S. Reiss, The paradox of software visualization, in: Proc. IEEE Vissoft, 59–63, 2005.

[4] S. Charters, N. Thomas, M. Munro, The end of the line for Software Visualisation?, in: Proc. IEEE Vissoft, 27–35, 2003.
[5] R. Koschke, Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, J. Soft. Maint. and Evol. 15 (2) (2003) 87–109.
[6] P. C. Wong, J. J. Thomas, Visual Analytics, IEEE CG&A 24 (5) (2004) 20–21.
[7] J. J. Thomas, K. A. Cook, Illuminating the Path: The Research and Development Agenda for Visual Analytics, National Visualization and Analytics Center, 2005.
[8] M. van den Brand, J. Heering, P. Klint, P. Olivier, Compiling language definitions: the ASF+SDF compiler, ACM TOPLAS 24 (4) (2002) 334–368.
[9] M. Lanza, *CodeCrawler* - Polymetric Views in Action, in: Proc. ASE, 394–395, 2004.
[10] F. Boerboom, A. Janssen, Fact Extraction, Querying and Visualization of Large C++ Code Bases, in: MSc thesis, Faculty of Math. and Computer Science, Eindhoven Univ. of Technology, 2006.
[11] T. Mens, S. Demeyer, Software Evolution, Springer, 2008.
[12] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer, 2006.
[13] N. Fenton, S. Pfleeger, Software Metrics: A Rigorous and Pracical Approach, Chapman & Hall, 1998.
[14] S. Diehl, Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software, Springer, 2007.
[15] H. Kienle, H. A. Müller, Requirements of Software Visualization Tools: A Literature Survey, in: Proc. IEEE Vissoft, 92–100, 2007.
[16] SolidSource BV, SolidSX, SolidSDD, SolidSTA, and SolidFX tool distributions, `www.solidsourceit.com`, 2010.
[17] SVCG, Scientific Visualization and Computer Graphics Group, Univ. of Groningen, Software Visualization and Analysis, `www.cs.rug.nl/svcg/SoftVis`, 2010.
[18] A. Telea, A. Maccari, C. Riva, An Open Toolkit for Prototyping Reverse Engineering Visualizations, in: Proc. Data Visualization (IEEE VisSym), IEEE, 67–75, 2002.
[19] G. Lommerse, F. Nossin, L. Voinea, A. Telea, The *Visual Code Navigator*: An Interactive Toolset for Source Code Investigation, in: Proc. InfoVis, IEEE, 24–31, 2005.
[20] L. Voinea, A. Telea, J. J. van Wijk, CVSscan: visualization of code evolution, in: Proc. ACM SOFTVIS, 47–56, 2005.
[21] L. Voinea, A. Telea, Visual Querying and Analysis of Large Software Repositories, Empirical Software Engineering 14 (3) (2009) 316–340.
[22] M. Termeer, C. Lange, A. Telea, M. Chaudron, Visual exploration of combined architectural and metric information, in: Proc. IEEE Vissoft, 21–26, 2005.
[23] S. Moreta, A. Telea, Multiscale Visualization of Dynamic Software Logs, in: Proc. EuroVis, 11–18, 2007.
[24] A. Telea, L. Voinea, Visual Software Analytics for the Build Optimization of Large-scale Software Systems, Computational Statistics (in print), see also `www.cs.rug.nl/~alext/PAPERS`.
[25] A. Telea, L. Voinea, An Interactive Reverse-Engineering Environment for Large-Scale C++ Code, in: Proc. ACM SOFTVIS, 67–76, 2008.
[26] R. Rao, S. Card, The table lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information, in: Proc. CHI, ACM, 222–230, 1994.
[27] A. Telea, Combining extended table lens and treemap techniques for visualizing tabular data, in: Proc. EuroVis, 51–58, 2006.
[28] D. Holten, Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data, in: Proc. IEEE InfoVis, 741–748, 2006.
[29] B. Shneiderman, Treemaps for space-constrained visualization of hierarchies, `www.cs.umd.edu/hcil/treemap-history`, 2010.
[30] A. Ludwig, Recoder Java analyzer, `recoder.sourceforge.net`, 2010.
[31] Redgate Inc., Reflector .NET API, `www.red-gate.com/products/reflector`, 2010.
[32] R. Ferenc, A. Beszédes, M. Tarkiainen, T. Gyimóthy, Columbus – Reverse Engineering Tool and Schema for C++, in: Proc. ICSM, IEEE, 172–181, 2002.
[33] LLVM Team, Clang C/C++ analyzer home page, `clang.llvm.org`, 2010.
[34] Y. Lin, R. C. Holt, A. J. Malton, Completeness of a Fact Extractor, in: Proc. WCRE, IEEE, 196–204, 2003.
[35] Bell Labs, CScope, cscope.sourceforge.net, 2007.
[36] T. Kamiya, CCfinder clone detector home page, `www.ccfinder.net`, 2010.
[37] VTK Team, The Visualization Toolkit (VTK) home page, `www.vtk.org`, 2010.
[38] A. Telea, O. Ersoy, Image-based Edge Bundles: Simplified Visualization of Large Graphs, Comp. Graph. Forum 29 (3) (2010) 65–74.
[39] A. Telea, L. Voinea, A Tool for Optimizing the Build Performance of Large Software Code Bases, in: Proc. IEEE CSMR, 153–156, 2008.
[40] L. Voinea, A. Telea, Case Study: Visual Analytics in Software Product Assessments, in: Proc. IEEE Vissoft, 57–45, 2009.
[41] M. Poppendieck, T. Poppendieck, Lean Software Development: An Agile Toolkit for Software Development Managers, Addison-Wesley, 2006.
[42] A. Telea, L. Voinea, O. Ersoy, Visual Analytics in Software Maintenance: Challenges and Opportunities, in: Proc. EuroVAST, Eurographics, 65–70, 2010.
[43] M. Sensalire, P. Ogao, A. Telea, Classifying desirable features of software visualization tools for corrective maintenance, in: Proc. ACM SOFTVIS, 87–90, 2008.
[44] R. Holt, A. Winter, A. Schurr, GXL: Towards a standard Exchange Format, in: Proc. WCRE, 162–171, 2000.
[45] S. Tichelaar, S. Ducasse, S. Demeyer, FAMIX and XMI, in: Proc. WCRE, 296–300, 2000.
[46] E. Korshunova, M. Petkovic, M. van den Brand, M. Mousavi, Cpp2XMI: Reverse Engineering for UML Class, Sequence and Activity Diagrams from C++ Source Code, in: Proc. WCRE, 297–298, 2006.
[47] Prefuse, The Prefuse Information Visualization Toolkit, `prefuse.org`, 2010.
[48] A. Lienhardt, A. Kuhn, O. Greevy, Rapid Prototyping of Visualizations using Mondrian, in: Proc. IEEE Vissoft, 67–70, 2007.
[49] M. van den Brand, S. Roubtsov, A. Serebrenik, SQuAVisiT: A Flexible Tool for Visual Software Analytics, in: Proc. CSMR, 331–332, 2009.
[50] A. Teyseyre, M. Campo, An Overview of 3D Software Visualization, IEEE TVCG 15 (1) (2009) 87–105.
[51] M. Sensalire, P. Ogao, A. Telea, Model-Based Analysis of Adoption Factors for Software Visualization Tools in Corrective Maintenance, Univ. of Groningen, the Netherlands, Tech. Report SVCG-RUG-10-2010, `www.cs.rug.nl/~alext/PAPERS/Sen10.pdf`, 2010.

# Building industry-ready tools: FAMA Framework & ADA[☆]

Pablo Trinidad, Carlos Müller, Jesús García-Galán, Antonio Ruiz-Cortés

*Dpto. Lenguajes y Sistemas Informáticos*
*ETS. Ingeniería Informática - Universidad de Sevilla*
*41012 Sevilla (Spain – España)*
{ptrinidad, cmuller, jegalan, aruiz}@us.es

**Abstract**

Developing good academic tools has become an art that forces researchers to achieve tasks they are not supposed to do. These include coding, web and logo designing, testing, etc. It compromises the quality of a product that differs from what the industry expects from it. In this paper we propose professionalising the tool development to build industry-ready products that are a middle term between academic and industrial products. We summarise many of the decisions we have made in the last three years to build two successful products of this kind, and that we think that may help other researchers their best in tool building.

*Keywords:* Automated Analysis, Software Product Line, SPL, Variability Models, Feature Models, Service Level Agreement, SLA

## 1. Context

### *1.1. Introduction*

One of the main objectives academic research must pursue is to turn ideas into industry-ready products. One of the most common ways software engineering researchers do this is by building software tools. However, many times academia only has resources to build just proof-of-concepts tools or prototypes that are not ready to be used by enterprises, but manage to demonstrate that the driving ideas are feasible and realisable into a product. When this is the case, prototypes are used to capture the interest of industry in academic research results. Alas, the time needed to transform a prototype into an industrial product is usually long enough to make businesses reconsider the real utility of this kind of prototypes. The industry expects mature products that can be incorporated into production as fast as possible. The academy's mind is on dissemination; the industry's mind is on *return on investment* (ROI).

In the spanish context, where most businesses are small or middle-sized, this distance is severe, since they are not usually prepared to assume the risk of incorporating the ideas behind a prototype into their products portfolio. It implies an opportunity loss that affects the dissemination of the research results and obviously the private incomes of our research structure.

[☆] This work has been partially supported by the European Commission (FEDER) and the Spanish Government under the CICYT project SETI (TIN2009–07366), and by the Andalusian Government under the project P07–TIC–2533.

We have observed that only academic tools are built when there is a lack of funding, and most of the times at the expense of researchers who invest too much time in tool building. In this scenario, researchers spend their time carrying out tasks they are not supposed to be assigned to; such as programming, testing, building a web page to disseminate their work or designing an interesting logo for the tool. Extra resources are not assigned to build tools because they increase expenditure, but letting researchers build tools is also an inefficient use of resources. In this scenario the inherited opportunity costs rocket since money is invested in good researchers but usually unexperienced tool-builders, rather than letting researchers investigate and allowing tool-builders develop tools.

The large distance between academic prototypes and industrial tools in quality terms is one of the main aspects that make companies refuse joint work with academia. We consider it necessary for academia to assume the costs of building better tools as efficient use of resources. As a projection of this reflection, for last three years we have been developing what we call industry-ready products that are a middle-term between academic tools and commercial products whose objectives are:

- Using tools to reduce the distance between academy and industry.
- Reducing the risk the companies assume when they are interested in research results.
- To allow a fast dissemination of research results in already existing industry-ready products.

In this paper we describe our experience from two points of view:

- Economic point of view: we have professionalised our tool-building activity so we have defined a business model that is briefly described in Section 2.
- Technological point of view: since we do not know our customers beforehand we have to build flexible products that could be adapted to the different companies who show some interest in our results. We show in Section 3 the main technologies that have helped to provide this flexibility and the methodologies that have been applied to lead our products to safe harbour.

### *1.2. Research Context*

Applied Software Engineering (ISA) Group is a research group composed of 22 researchers. Our research focuses on three main lines: *Software Product lines* (SPL) [1], software methodologies and *services oriented architectures* (SOA). Each line is independent of the others and there are only sporadic collaborations among researchers in different lines. For each line, several tool prototypes are built to demonstrate the realisation of the different advances and contributions. Within the wide set of prototypes we have built, there is a subset whose objective is analysing models of any kind: *Feature Models* (FMs)[2] in SPL, *Service-Level Agreements* (SLAs)[3, 4] in SOA, BPMN models in methodologies, etc.

In 2007, we decided to transform *SPLReasoner*, a promising prototype to analyse FMs, to *FAMA Framework* (FAMA FW), an industry-ready product. In 2009, we could say that this model worked for FAMA FW so we decided to replicate the process to transform SLAxplainer [5, 6] into ADA [7], both tools to analyse SLAs. In ADA, we have applied not only the economical aspects that led FAMA FW to be a success story, but also most of the technological issues since their context is close enough to reuse technologies and methodologies. Next we describe what we expect from an analysis tool in general and in FAMA FW and ADA in particular.

### *1.3. The products: Analysis Tools*

Generally speaking, an analysis tool is used to extract relevant information from a model. To achieve it, the tool transforms a model into different artificial intelligence paradigms such as constraint programming, satisfaction problems or genetic algorithms. Users can query for information by means of different analysis operations and the tool selects the most suitable technique to solve the analysis problem, having the best possible time-to-response.

In 2007, SPLReasoner had become an ongoing prototype in which the SPL community had shown much interest. We decided to take it to the next level building FAMA FW, an industry-ready tool that should fulfil two requirements:

- We will develop it as if it were an industrial product rather than an academic product. The main difference arises in how seriously we will take the development process and the support to users and third-party developers. There is an economic investment need, so business and dissemination models are defined to justify each cent that is invested in the project.
- The product must be customisable since there are many potential customers. This means that we will have a product per customer, thus transforming the product into an SPL. The methodologies and technologies must be adapted to support an SPL.

There were no new functional requirements added to FAMA FW, only non-functional or quality requirements that increase the flexibility of the product. These features did not depend on FAMA FW particularities so, ADA was built to fulfil the same requirements. The architecture of the product would be completely different so both products were built from scratch and only some isolated pieces of code could be reused from the original prototype. Human resources were assigned to the project and development methodologies were established to manage the development team.

### *1.4. Objectives*

At date, the practises that we describe in this paper are part of our internal procedures to build and maintain software products of near-industry quality; so the distance between research results and the industry is lessened. Our main objective is for the reader to see our story as a reflex in their context and to consider applying some of the ideas described in this paper to transform prototypes into industry-ready products.

We have divided our experience in two parts: a first part in Section 2 that explores the economic aspects that have been considered during FAMA FW and ADA's lifetime. These are the interaction with the industry, human resources hiring, how to improve product dissemination and which are the risks to be taken into account. A second part in Section 3 focuses in describing the development methodology and how the architecture and the chosen technologies have helped to manage the evolution of the product. To conclude we have built a *post-mortem* report in Section 4 that summarises the best and worst practises that we have detected during the development of the tools under study.

## 2. Business Model

In this section we describe some organisational aspects that caused a reorganisation of our structures and intense decision processes that took up a large amount of time and have to be taken into account whenever an industry-ready tool is built.

*2.1. An iterative process to build industry-ready products*

***Why do we build tools?***. When we look to define a better way to build tools, it is mandatory to analyse the reasons why academia builds tools. We build tools to transform research ideas into a tangible product. If a company shows interest in the product, the product generates income. This income allows academia to keep on researching and the research continues to produce ideas that are incorporated to existing products, or in itself leads to the development of new products.

And why does a company pay for a product or finance a research topic? Because they foresee its transformation into a commercial product and they expect to obtain ROI from it.

There is also another scenario that must be considered; the creation of spin-off companies originating from these products. Mature products can become important assets in this kind of companies which are created to industrially exploit research results.

***Why does academia builds prototypes instead of building commercial products?***. A prototype is the less expensive and risky procedure to validate clipboard results. It reduces the time-to-validation by a proof of concept. Although they work for researchers, this is not the appropriate form to present the product to the industry. The industry expects a better quality product so researchers usually invest time in improving the quality of a prototype.

***How is a research prototype built?***. Usually a researcher is the only developer. The tool is generally created as part of the ideas maturing process. Since transmitting vague ideas to developers foreign to the problem may delay the project or even make it fail, the researcher is the one in charge of developing the prototype. Although this way of working is valid when first creating a prototype, extending the development in time may lead to a misuse of researchers time.

***Where are the limits with prototyping?***. This is one of the most important questions to be answered by each researcher that develops a prototype. In our case, we have found that the correct answer for us is *'When you can estimate the time and cost to build an industry-ready product, the core requirements and the target market'*. The point of this is that a researcher does not spend more time in development tasks than what is necessary to confirm ideas. What a research group needs is to collect all the available information to evaluate if it is worthy or not to assign resources to build an industrial-strength tool from a prototype.

***May a research prototype become a commercial product?***. The general answer is no. A prototype easily becomes a legacy system. Usability, extensibility and adaptability are quality attributes that are barely considered as initial requirements when a researcher decides to build a prototype. This transforms a prototype into a hard-to-maintain product. We need flexible architectures that can adapt a product to customer needs and incorporate new research results, and a prototype is not usually built following these two objectives.

***How do we build an industry-ready tool?***. We have defined a process to manage the development of tools that goes from the generation of research results to the dissemination of results to companies. The process is divided in four main phases:

1. **Prototype initiation**: a prototype is initiated under the researcher's consideration without constraints of any kind. It is important that the researcher does not try to build the definitive product; otherwise he will spend time on work that will be quickly discarded. Once a prototype is mature enough, it passes to the next phase.

2. **Prototype evaluation**: a research group committee evaluates how promising a prototype is based on the reports submitted by researchers, which have to analyse the development costs, product features and target market among other issues. If a prototype passes this phase, resources are assigned and the researcher is appointed leader of the product development.
3. **Prototype design**: a reference architecture is defined. It must be as flexible as possible so it supports any further extension or change in products requirements. Product leader and hired developers are involved in this phase.
4. **Product lifecycle**: whilst sufficient resources are assigned, products are developed following an iterative process. The lifecycle is carried out in three sub-phases:

   - Transfer: ideas are incorporated to the reference architecture as a new feature. The hired human resources are expected to get involved mainly in this phase. This phase ends when the feature is implemented, tested and correctly documented.
   - Tracking: the committee evaluates the product by means of the feedback received from companies that have shown interest in the product. Based on this feedback, future ideas to be investigated are chosen. The resources assignment is supervised and changed if needed.
   - Research: researchers produce new ideas. They are normally validated by the community in workshops, conferences or journals. When an idea is mature enough, *transfer* phase starts.

   Each of these sub-phases are executed by different human profiles: researchers, committee members and developers. This work distribution allows the sub-phases to overlap.

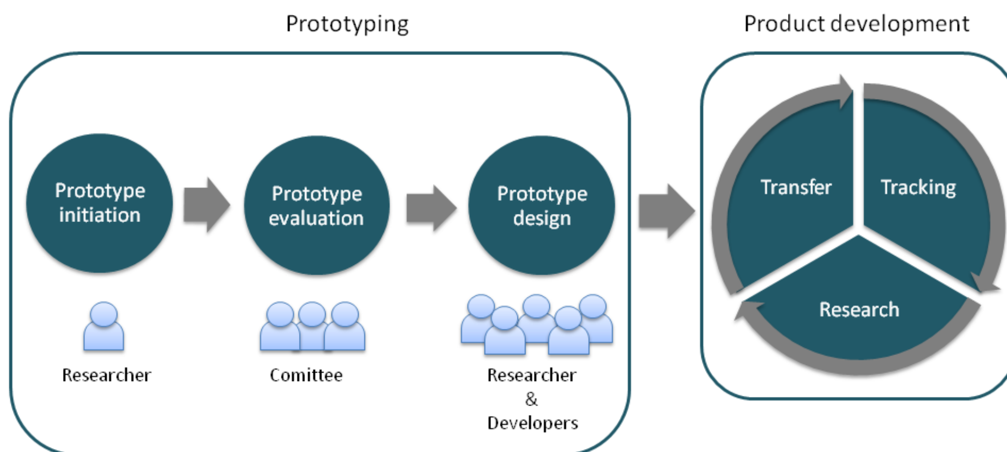



Figure 1: Our Business process model

***How many iterations are needed to produce an industry-ready product?***. We have to take into account that this form of building products is like creating a product whose market is unknown. When we have a customer, requirements are clearer. However, when there exist many potential customers but none of them acts as real customer, there is a tendency to be involved in a never-ending project. If a company shows interest in a product, the iterative process may be repeated indefinitely. If after given several iterations, no interest is shown, it is better to freeze the project until new opportunities arise so that resources are not wasted. In spite of this, the more iterations are fulfilled, the clearer the companies vision of the product. It is the commitee's responsibility to envision future interests and decide which risks are worth taking and which are not.

### *2.2. Human Resources*

With the above approach, we try to manage the product development as if we were a software development company. Thus, we have defined several roles that are played by our researchers and the hired human resources such as project leaders, developers, testers, etc. We have stated that researchers act as project leaders but, who plays the remaining roles?. It is not a secret that researchers commonly play roles they must not play such as developer, tester, designer, web-master, etc. We consider this is a waste of resources since researchers salary is usually (and hopefully) higher than developers. We attempt to hire different workers whose professional profile fits into the job they are assigned.

When hiring people, a new problem arises: carrying out a selection process. We have delegated this task to the project leaders, however several factors make us wonder if this decision is the best choice:

- On the one hand, the University has plenty of motivated and well-prepared candidates. In our case, most of the researchers are lecturers in the University so they have probably taught some matters to the candidates, making it easier to evaluate their personal and technical abilities. This information is very valuable in the selection process.
- On the other hand, selecting people is not one of the responsibilities a researcher must assume. Furthermore, it is interesting to differentiate candidates who want to be part of a big software company from candidates expecting a professional career in academia. We want to avoid a developer leaving an ongoing project since it would imply a new selection process and probably the loss of knowledge. This makes the selection process harder and it usually takes longer than expected.

After 3 years following this procedure, we have had good and bad experiences. We have organised selection processes lead by the project leaders. Although the project leader is who best knows the technical profile a developer or a tester has to satisfy, there are other psychological and emotional factors involved in selecting the right candidate. Technical skills are as important as personal abilities. This is one of the reasons that makes us wonder if selection processes should be outsourced so there is more confidence in selecting the most decisive candidate.

### *2.3. Dissemination*

Tools dissemination is a task that should be carefully performed while the project takes place. We explain as follows the most important issues of our *marketing* and *communication plans* in order to define the different actions related to the dissemination of our analysis tools.

#### 2.3.1. *Marketing Plan*

A marketing plan defines the products we aim to build, the policies to regulate its consumption, and its target market.

The basic policies needed to regulate the consumption of our developed analysis tools are the *price* and *usage policies*.

- Price policies: our goal is to capture investment; that means to obtain as much users as possible for our products. Users are not asked for a fee to use our tools. We decided to offer them as open-source tools, yet some kind of tool supporting such as tool customisation or integration would require an economic compensation.
- Usage policies: We set LGPLv3 licence [8] to our products. We decided to use this licence since it allows everyone to use our tools, even in commercial products, but a citation to the owners is mandatory.

We established as target market for our products companies and research institutions that have shown some interest in analysing models of any kind, specifically FMs and SLAs. We would offer our analysis tools and customising services to them. At this point we had to set a communication plan that considers how to reach the target market.

#### 2.3.2. *Communication Plan*

Our communication plan's objective is two-fold; we want to be equally appealing to the two entities of our target market: private companies and research institutions.

To achieve such purpose we had to ask ourselves: What do private companies and research institutions expect from us?. While the former usually have the goal of getting a commercial product in order to obtain ROI; the latter commonly want to work together to obtain results in terms of research and tools integration. Therefore, we have to consider the kind of entity in the definition of our communication plan as follows:

- Dissemination releases for companies and research institutions:
  - *Advertising*: we have created web portals as direct communication channels[1]. Before releasing any versions of our products we designed logos and corporate images that identify the products in any publication relating them. Another kind of advertising is with posters and tool demonstrations sessions of conferences and commercial events.
  - *Technical documentation*: customers need to know how to use and extend our tools by means of: (1) *technical reports* published in the respective web portals. Such technical reports reduce the time to learn to develop an extension of our tools or to integrate our tool into third–party tools. Usually, the technical reports are published on–demand when an entity shows interest so our effort is not wasted. (2) *source documentation* by means of Javadoc [9] or Doxygen [10] tools. Doxygen allows introducing LaTeX pieces of text/images and producing a final documentation richer than the one produced by Javadoc. We also include source code documentation.

[1] FAMA portal is available at `http://www.isa.us.es/fama` and ADA portal is available at `http://www.isa.us.es/ada`

- *Scheduling*: periodically, private companies go to commercial events and research institutions go to conferences. Once these forums are identified, we attend in order to show our products.
- *Predefined forms*: some legal documents such as agreements and compliance forms and usage certificates are defined. These documents are necessary to validate the use of our tools by other entities and to guarantee the LGPLv3 compliance. Therefore, when an entity starts using a product of ours, they will be asked to request a signed copy of such documents.
- *Periodical affiliation*: to keep the affiliation of research institutes and companies they are periodically sent quality assurance forms, satisfaction forms, etc.

- Dissemination releases only for research institutions:
  - *Research-oriented dissemination*: usually the communication of research tools is not mass marketing, so a much more selective marketing activity is needed. Neither leaflets, nor mailings are needed. Conferences and workshops are the most suitable events for tool demonstrations. In this case, scientific papers and journals are a first–term documentation, but they do not usually describe the internals of tools. Specially, when the research contribution is the structure and design of the tool instead of a new algorithm. Technical reports cover this gap so the insides of tools are described without constraints of any kind.

- Dissemination releases only for companies:
  - *Company-oriented dissemination*: Any research group grows surrounded by local, national and worldwide companies who are potential consumers of our products. Research projects usually include some technology partners that are interested in knowing at first hand the results of the projects. This contact usually produce constant meetings between researchers and companies that may lead to projects where our products play a key role.

*2.4. Risk Analysis*

A *risk* is any event that implies a delay in time or a deviation in expenditure. We have identified the following risks during industry-ready product development:

1. *a developer leaves*. It is a well-known risk in any development process. Avoiding it is complex, but we attempt to reduce its probability trying to offer our technical staff obtainable goals to prepare professional careers accordingly. If it cannot be avoided, it is important to minimise the impact of a developer leaving. It implies a constant focus on source code documentation and reporting.
2. *a product is not interesting for the community*. When this happens, it affects the cost assumed to date. Therefore, we establish the iterations of process depicted in Section 2.1 to determine as early as possible if the target market is interested in a product or not.
3. *A researcher has to play other roles*. It usually happens when the project has been assigned insufficient human resources. A thorough project scheduling helps to avoid these situations, anticipating the needed human resources.

## 3. Development model

In this section we describe the software development methodology followed in the development of our analysis tools. We also detail the technology incorporated in our products and the reference architectures.

### *3.1. Methodologies*

Due to the particular characteristics of our products, a methodology to develop our industry-ready products should support: (1) changing requirements, because requirements become clearer when the results can be observed, and (2) a straightforward and continuous communication between researchers and developers to ease the incorporation of research results into the existing products.

To cover these needs, we chose agile methodologies [11] which are a set of methodologies based on teamwork and adaptation to changes to provide a rapid high quality software product. These methodologies requires small teams, frequent and straightforward communication as periodic meetings to monitor the progress of the tool, a short time planning and the know-how developers reuse from similar projects to take advantage of their experience. The opposite of heavyweight methodologies, agile methodologies are more incremental, iterative, present short iterations, and requires less documentation; only the essentials, so third-party developers can understand how to extend the tool, users know how to use the product and the know-how remains inside the organisation.

We have specifically used the following agile methodologies to develop our products:

- *Feature-Driven Development* (FDD). Firstly, groups of features, which represent user visible characteristics, are identified. Later, such features are classified in core and satellite features for a priority assignment. A core feature is mandatory for any potential customer; a satellite feature is only relevant for a subset of customers. In an iteration, a subset of features is selected, designed, developed and tested starting with the core features and following the priority order. FDD allows the separation of different satellite projects, each considered with satellite features.

- *Test-Driven Development* (TDD). Before developing or upgrading a feature, one or more test-cases are designed, implemented and executed. Whenever new source code is produced or existing one changes, code is tested. Finally we refactor the source code to grant its quality. In addition, before building a distribution for a product, test-cases are executed to detect possible errors introduced by changes. It increases the quality, reduces the time-to-failure, and accelerate the development since time to repair errors is reduced. TDD is used in our projects in conjunction with FDD to increase the confidence in the feature development correctness.

The experience on using these methodologies has been very satisfactory. On FAMA FW, several versions of the tool have been released with significant functionality upgradings and the amount of detected errors has been drastically reduced. On ADA, the experience on developing and managing FAMA FW helped us to improve the development process. Thus, we built the first version of ADA which covered a complete iteration in all the development phases in three months.

*3.2. Software Architecture*

Before defining an architecture we need to discuss the common aspects of analysis tools in general. Basically, an analysis tool works with a *model* to be analysed, several *analysis operations*, and one or more *reasoners* that are able to solve those operations. The main goal is extracting information from the models using the analysis operations. Using this approach we may characterise a flexible analysis tool as the one that is able to support:

- Different kinds of models (a.k.a. metamodels) and file formats to store and retrieve them.
- Different analysis operations.
- Several alternatives to solve an analysis operation.

FAMA FW and ADA have a SPL-based design. This means that instead of building a product, we pretend to build a flexible production platform that is able to produce different kinds of products depending on the customer needs. It is very valuable for our purposes since we are unable to determine the companies that are going to use our products.

SPL is a building concept rather than a bunch of methodologies, so specific procedures have to be defined for each kind of project. One of the ways to implement a SPL is by a component-based architecture [12, 13]. A component-based architecture emphasises on developing functionality in independent components, integrating them later to provide a complete functionality. It allows dividing the tool in many small-sized projects and reducing the coupling among them. It permits parallelising the development of these components so development time can be reduced. Among the advantages of using a component-based architecture we remark the following:

1. Modularity: coupling among components is reduced to minimum since each of them is specific enough to perform a feature without needing a direct coupling with other components. It can only be achieved when the nature of the problem allows this kind of division, as is the case for our products.
2. Extensibility: new functionalities are easily added registering a new component in the reference architecture.
3. Integrability: due to the reduced coupling, communication lines between components is minimum which eases the integration process and the substitution of components.

In our component-based architecture, we have identified three component layers as depicted in Figures 2 and 3:

1. Core: a core is a subset of components that must be in any product. They are the first pieces to be defined in any tool. Core components define basic functionalities, public interface for the final users, and load and use extensions.
2. Extensions: an extension is an implementation of one or more features that are pre-defined by core. Several extensions can be available offering the same feature so the core is in charge of selecting the most suitable extension to carry out the job. The extensions we have identified for FAMA and ADA are:
   - Metamodels to be analysed.
   - Analysis operations.
   - Reasoners to solve the analysis operations.
   - Selection criteria to choose the better reasoner to solve an analysis operation.

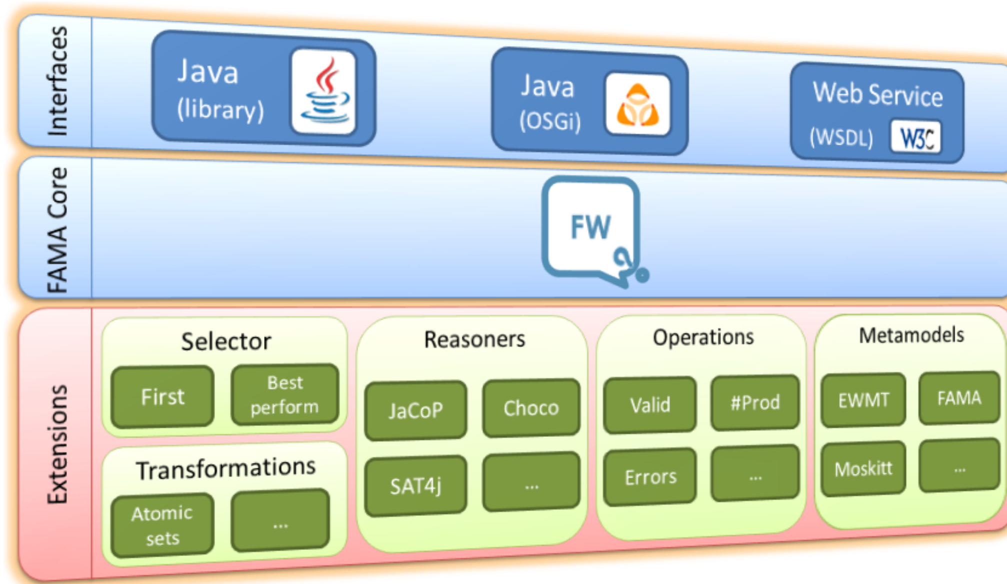



Figure 2: Architecture of FaMa framework

- Transformations between models.

3. Interfaces: they are the facade that allows third-party entities to use our products. They are defined as independent satellite projects that use the API of the analysis tool, considering tool API the core and extensions, and give us new functionality. A typical example can be a GUI, that consumes the API of the tool and makes the human use of the tool easier than the programmatic use. An example of this kind of interface is available for ADA, specifically a rich internet application called *ADA-FrontEnd* that is accessible at *try it online!* section of ADA portal (`http://www.isa.us.es/ada`).

In both tools, FAMA FW and ADA, a three-layer architecture (see Figures 2 and 3) is designed to support several *variation points*. In our context, a variation point is any extension component.

From a developer's point of view, the tool becomes a framework, since any new extension that is developed by third-parties can be integrated in the architecture and the core is in charge of consuming the extension. From an end-user's point of view, those variation points are transparent to the user since it is the core that is in charge of choosing the extensions needed without any kind of user interaction.

The architecture permits dividing the product development into several sub-projects each of them involving an extension. It eases the community collaboration and the hiring of new developers for our development team who can start developing satellite projects without affecting core functionality.

However, the adoption of this architecture involves a double challenge: (1) the integration between independent components requires a technological solution, such as a tool or framework to support it; and (2) the management of dependencies between core and extensions. The following Section 3.3 tackles these challenges with some technology decisions.

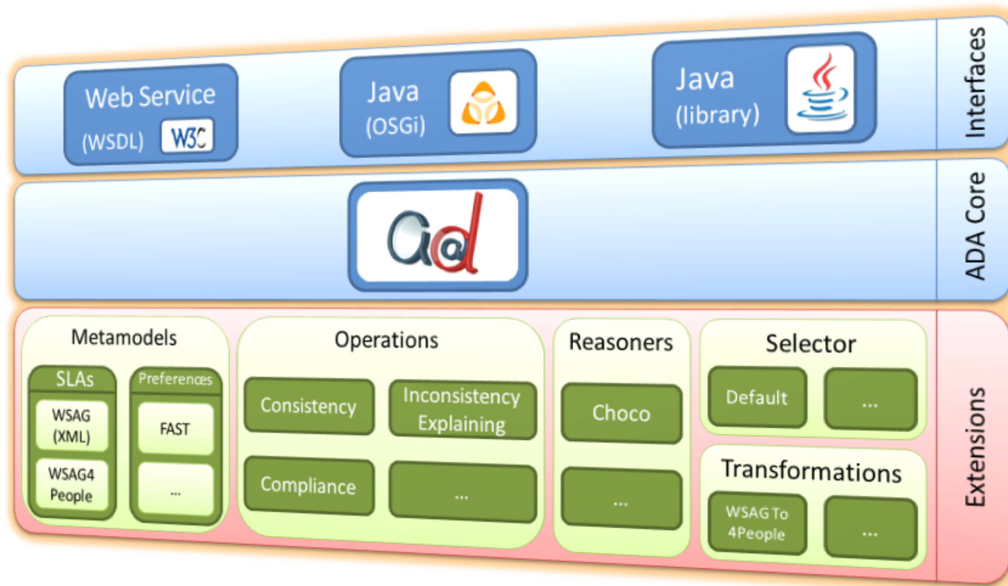



Figure 3: Architecture of ADA framework

### 3.3. Technologies

Making the right decisions regarding technology is key to support product evolution. Any wrong decision may lead to an important economic loss.

One of the most important choices to develop software is the programming language to use. A stable and mature language, with the right paradigm is needed to minimise risks. If we want to involve a community behind a product, we should choose a general purpose programming language with many open source projects. These projects implies to have several free tools available that can be incorporated into our products. We have chosen Java [14] because it is widely used and has a large open source community behind, and many high-quality tools.

In previous Section 3.2, we mentioned two challenges in architecture designing: integration between components and dependencies management. OSGi [15], Maven [16] and Subversion [17] have been the chosen tools that help us with these problems as detailed in the following:

1. *OSGi* is a framework specification that provides a container to deploy, run and integrate components (a.k.a. *bundles*) for this framework. It has several implementations, such as Equinox [18], Felix [19] or Knoplerfish [20]. OSGi allows the integration between OSGi compliant tools, and it offers some interesting plug-in tools. One of them is Distributed OSGi (DOSGi) [21] which publishes OSGi bundles as web services providing a WSDL specification [22][2].
   ADA and FAMA FW are OSGi-compliant. FAMA FW is currently being used by Moskitt Feature Modeller [23] thanks to OSGi framework, and ADA public interface can be used as a web service through its WSDL specification.

[2] *DOSGi* was used to generate the *WSDL* facade of commented ADA interface *ADA-FrontEnd*

2. *Maven* is a software tool for the project management and for automating tasks on java projects. For FAMA FW and ADA, Maven manages the dependencies between core, extensions and satellite projects, OSGi metadata, binaries, and tests execution before packaging.
3. *Subversion* is a popular version control system to manage changes on source code, resources and documentation. Our subversion repositories comprise three folders: (1) *trunk* for the current development, (2) *branches* for separated lines of development, and (3) *tags* to keep snapshots of the repository at a concrete date or milestone.

## 4. Conclusions and Postmortem Report

With this experience we have learned that we, the researchers in academia have to put ourselves in industry's shoes so our prototypes are closer to what the industry expects. As of today, more than 20 companies and research institutions are using, adapting or are interested in using FAMA FW and we expect ADA achieves the same goal in the next months. During this process there have been right and wrong choices. To analyse them we have realised a post-mortem report which summarises those practices that we will keep in the future and that we recommend others to do. Besides, we have committed errors that we will try not to repeat in the future and are also part of this report.

### *4.1. How to continue doing what was done Right*

1. LGPLv3 license is the best option.
2. The components architecture has reduced the bugs in the products, moreover in those that have been affected by many iterations.
3. The business model used in FAMA FW has been validated in the ADA project. The human resources reuse between the two projects has been especially interesting.
4. The dissemination model is effective enough as shown by the high number of companies and research institutions interested in FAMA and ADA.

### *4.2. How to correct what Needs Improvement*

1. We have not focused our efforts in providing much documentation to end-users and third-party developers. We have had to invest much time in documenting after several iterations since we have been asked for more and better documentation. We have solved it by adding a new procedure to FDD that requires documenting before ending the development of a feature or extension.
2. Some times our selection processes were massive and they took longer than expected. We will find a solution for this in the future.
3. At the beginning we prepared a detailed planning for the whole project and we used a tasks tracker to monitor if tasks were on schedule. However, we had to redo the planning due to the changing requirements. Therefore, the detailed planning should define short-term goals within a month. The long-terms goals should be defined in a more relaxed planning.


[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison–Wesley, 2001.

[2] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature–Oriented Domain Analysis (FODA) Feasibility Study, Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (Nov. 1990).

[3] L. Lewis, Managing Business and Service Networks, Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[4] D. Verma, Supporting Service Level Agreements on IP Networks, Macmillan Technical Publishing, 1999.

[5] C. Müller, A. Ruiz-Cortés, M. Resinas, An Initial Approach to Explaining SLA Inconsistencies, in: Proc. of the $6^{th}$ Int. Conf. on Service-Oriented Computing (ICSOC), Vol. 5364 of LNCS, Springer Verlag, Sydney, Australia, 2008, pp. 394–406.

[6] C. Müller, M. Resinas, A. Ruiz-Cortés, Explaining the Non-Compliance between Templates and Agreement Offers in WS-Agreement*, in: Proc. of the $7^{th}$ International Conference on Service Oriented Computing (ICSOC), Vol. 5900 of LNCS, Springer Verlag, Sweden, Stockholm, 2009, pp. 237–252.

[7] C. Müller, A. Durán, M. Resinas, A. Ruiz-Cortés, O. Martín-Díaz, Experiences from building a ws–agreement document analyzer tool (including use cases in ws–agreement and wsag4people), Tech. Rep. ISA-10-TR-03, ISA Research Group, `http://www.isa.us.es/modules/publications/getPdf.php?idPublication=322` (Jul 2010).

[8] Free Software Foundation (FSF) Inc., Gnu lesser general public licence version 3, http://www.gnu.org/copyleft/lesser.html.

[9] Oracle–Sun Developer Network (SDN)., Javadoc tool, http://java.sun.com/j2se/javadoc/.

[10] Dimitri van Heesch et al., Doxygen -Dox(Document) Gen(Generator)-, http://www.doxygen.org.

[11] M. Fowler, J. Highsmith, The agile manifesto, In Software Development, Issue on Agile Methodologies (August 2001).

[12] D. Coppit, K. Sullivan, Multiple mass-market applications as components, in: Proc. of the 2000 International Conference on Software Engineering, 2000, pp. 273–282.

[13] H. Kienle, Component-based tool development, in: Frontiers of Software Maintenance, 2008. FoSM 2008., 2008, pp. 87–98.

[14] Sun Microsystems, Java, http://java.sun.com.

[15] OSGi Alliance, Osgi, www.osgi.org.

[16] Apache Software Foundation, Apache maven, http://maven.apache.org.

[17] Apache Software Foundation, Subversion, http://subversion.apache.org.

[18] Eclipse, Equinox, www.eclipse.org/equinox.

[19] Apache Software Foundation, Apache felix, http://felix.apache.org.

[20] Makewave, Knopflerfish, www.knopflerfish.org.

[21] Apache Software Foundation, Distributed osgi, http://cxf.apache.org/distributed-osgi.html.

[22] World Wide Web Consortium, Web Services Description Language (WSDL) 1.1, `http://www.w3.org/TR/wsdl` (2001).

[23] Conselleria de Infraestructuras y transportes, Moskitt, http://www.moskitt.org/cas/moskitt0.

# Developing A Generic Debugger for Advanced-Dispatching Languages☆

Haihan Yin[a], Christoph Bockisch[a],

[a] *Software Engineering group, University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands*

**Abstract**

Programming-language research has introduced a considerable number of advanced-dispatching mechanisms in order to improve modularity. Advanced-dispatching mechanisms allow changing the behavior of a function without modifying their call sites and thus make the local behavior of code less comprehensible. Debuggers are tools, thus needed, which can help a developer to comprehend program behavior but current debuggers do not provide inspection of advanced-dispatching-related language constructs. In this paper, we present a debugger which extends a traditional Java debugger with the ability of debugging an advanced-dispatching language constructs and a user interface for inspecting this.

*Keywords:*
*2000 MSC:* 68N15 Programming Languages,
*2000 MSC:* 68N19 Other programming techniques,
*2000 MSC:* 68N20 Compilers and interpreters, Debugger, Advanced-dispatching, Eclipse plug-in

## 1. Introduction

To improve the modularity of source code, a considerable number of new programming-language mechanisms has been developed that are based on manipulating dispatch, e.g., of method calls. Examples are multiple [1] and predicate dispatching [2], pointcut-advice [3]—a particular flavor of aspect-oriented programming (AOP)—, or context-oriented programming [4]. Because they are beyond the traditional *receiver-type polymorphism* dispatch mechanism, we call these new mechanisms *advanced-dispatching* (AD).

The majority of newly developed languages adds advanced-dispatching concepts to an already existing, mainstream language like Java or the .NET languages, which is generally called the *base language*. The new mechanisms have the potential to increase the modularity of program code compared to code written in the base language. However, using the new language mechanisms is not well-supported by tools. Thus their usage may hamper software development even in spite of their potential of improving the code quality.

☆This work is partly funded by a CSC Scholarship (No.2008613009).

*Email addresses:* h.yin@student.utwente.nl (Haihan Yin), c.m.bockisch@cs.utwente.nl (Christoph Bockisch)

The term *dispatching* refers to binding functionality to the execution of certain instructions, so-called *dispatch sites*, at runtime, thereby choosing from different alternatives that are applicable in different states of the program execution. An example of conventional dispatch is the invocation of a virtual method in object-oriented languages: The invocation is the dispatch site and the alternative functionalities are the different implementations of the method in the type hierarchy; the runtime state on which the dispatch depends is the dynamic type of the receiver object. A detailed discussion of the approach can be found in [5][1].

With *advanced-dispatching* we refer to language mechanisms that go beyond this traditional receiver-type polymorphism. What makes the dispatching advanced in these cases is that a dispatch can consider additional and more complex runtime states, and that functionality can be composed in various ways.

In our work, we have especially investigated the advanced-dispatching languages JPred [6], MultiJava [7], AspectJ [8], Compose* [9], CaesarJ [10], JAsCo [11], and ConSpec [12]. While high-quality tools, such as the AspectJ Development Tools [13], exist to visualize the static structure of programs written in these languages, little to no support is provided related to dynamic language features. Lack of supporting tools significantly hampers the popularity of otherwise valuable new programming languages, thus dedicated tool support is required. In a previous position paper [14] we have already outlined this claim as well as a an implementation of language independent tools for advanced-dispatching languages based on the ALIA4J architecture.

This paper specifically focuses on a generic debugger tool which is aware of advanced-dispatching concepts and allows to inspect such program elements at runtime. This debugger is realized based on a language-independent intermediate representation and execution environment of advanced dispatching and can therefore support developers using all the above-mentioned languages. Instead of implementing it from scratch, we chose to extend the Java debugger of Eclipse with AD-specific features. Figure 1 shows how our debugger graphically represents a dispatch at which the execution is currently suspended. The provided views are discussed in detail in section 5.

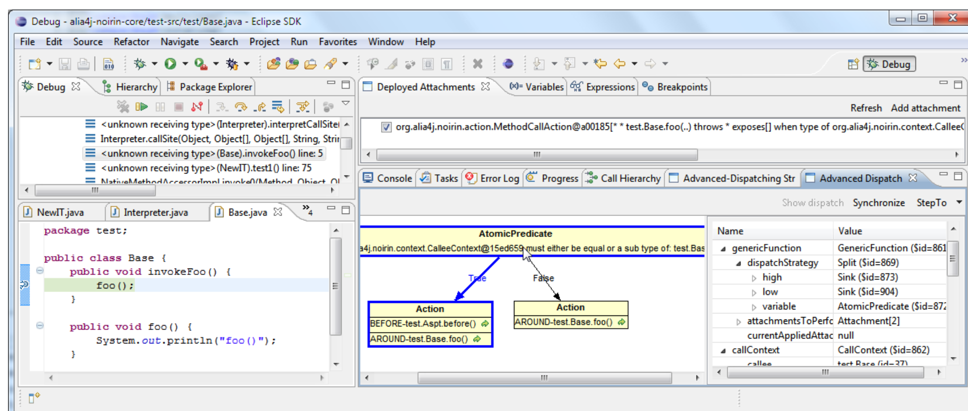



Figure 1: A snapshot of the debugger for advanced-dispatching programs

[1]Some details presented in [5] are outdated, but it may nevertheless act as an introduction to the basic concepts.

The remainder of this paper is structured as follows. In section 2, we present a deeper analysis of the problems when debugging advanced-dispatching programs as well as features we claim are required from a good debugger for such programs. Section 3 outlines the approach of our work, followed by a section presenting the required technical background (section 4). Next we discuss the three corner stones of our debugger, namely the user interface (section 5), the extension to the execution environment (section 6), and the extended debug model (section 7). This paper closes by presenting an example of debugging an AspectJ program with the developed debugger in section 9, related work in section 10, and some conclusions and future work in section 11.

## 2. Problem Analysis and Goals

The lack of tool support for advanced-dispatching languages is based in the technique typically applied when implementing such languages: in all investigated language implementations, a compiler translates source code from the new language to code—most frequently an intermediate representation like Java bytecode [15] or the .NET common intermediate language [16]—of the base language. After the compilation all source code is mapped to the same intermediate language whose abstractions reflect the module concepts of the base language, but not those of the new language. This compilation strategy is often called *weaving* and may entail that instructions are removed, merged, duplicated, reordered, or interleaved. This is discussed, for instance, by Avgustinov et al. [17] in the context of the AspectBench Compiler for the AspectJ language.

As the new language concepts are compiled to conventional code, the tooling, e.g., the debugger, of the base language can also be used for compiled programs written in the new languages. However, as it is not aware of the new language concepts, the base tooling may not always be adequate. Take the Java debugger for example: When the debugged program is intercepted, the Java debugger offers the possibility to locate the source code that was compiled to the Java bytecode instruction whose execution was intercepted. This feature is supported by the Java bytecode format that stores for every bytecode file the name of the source-code file from which it was compiled, as well as a mapping from instructions to lines in the source file.

When this debugger is used, e.g., for AspectJ programs, problems like the inability of locating source code are easily encountered because programmers debug the transformed, woven code. After the weaving phase, the assumption—which is manifested in the Java bytecode format—that all code of a class is compiled from a single source file may no longer hold.

As a starting point for our investigations, we use Eaddy et al.'s [18] categorization of the debugging problems for aspect-oriented programs into the *code-location problem* and the *data-value problem*. Since aspect-orientation is one special case of advanced-dispatching, we generalize these problems and conclude that existing debuggers are not sufficient for AD languages mainly for the following two reasons:

**Code-location problem** One important purpose of debugging is locating errors in the source code. As explained previously, the compiled code may undergo a series of transformations during which the source-location information is not maintained by current compilers. This lack of traceability causes the debugger to show no or the wrong source code, or to show compiled and woven *intermediate code* instead of the original *source code*.

**Data-value problem** Dispatch results in the execution of one of multiple alternative functionalities; which alternative is chosen is evaluated according to the runtime context in which

dispatch site is executed. While a conventional debugger will eventually show which functionality is executed, it is unable to present to the developer the reasoning behind choosing this functionality. Take debugging an AspectJ program in Eclipse for example: the debugger does not provide aspect-related information like runtime states accessed during dispatch, e.g., in terms of dynamic pointcut designators.

Inspired by the work of Eaddy et al. and by our own observations when using advanced-dispatching languages, we concentrate on solving the data-value problem and formulate the following debugger capabilities that are desirable for an advanced-dispatching debugger:

1. Inspection of runtime state of the executing AD program.
2. Inspection of program composition and control flow.
3. Inspection of the evaluation result of functions over the runtime state to select alternative meanings.
4. Description the relationship between AD entities.
5. Deployment and undeployment AD features at runtime.

## 3. A Debugger for Advanced-Dispatching Languages

To enable debugging, the debugger front-end must communicate with execution of the debugged program. Furthermore, the source code must be traceable, i.e., it must be possible to deduce the source code that has lead to a certain execution. Therefore, we base the debugger presented here on our previous work [19, 5], the ALIA4J approach for implementing advanced-dispatching languages. It provides a uniform representation of programs written in different advanced-dispatching languages in order to enable the reuse of implementations between these languages, and it embodies the execution semantics for advanced-dispatching in a language-independent way. Furthermore, its representation of advanced-dispatching stays first-class during the execution. Therefore, we present a debugger for advanced-dispatching languages, which is based on the ALIA4J approach, in order to improve the tooling landscape for multiple existing and future advanced-dispatching languages at once. Basically, our work will allow the developer to debug the ALIA4J representation of advanced-dispatching instead of woven bytecode as in traditional approaches. The ALIA4J representation is much closer to the original source code than the woven bytecode, and preserves AD-related debug information, thus increasing the traceability.

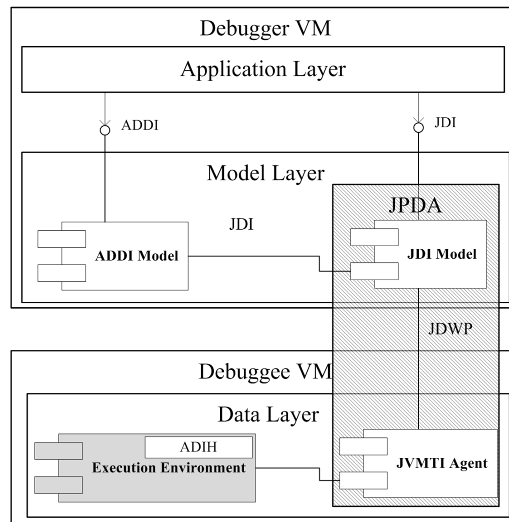



Figure 2: Structure of the AD language Debugger Architecture

We have developed the *AD language Debugger Architecture* (ADDA) as an extension to the *Java Platform Debugger Architecture* (JPDA) [20] based on ALIA4J's representation of advanced-dispatching. Figure 2 shows the overall structure of the ADDA. The *Execution Environment* component is extended with functionalities supporting debugging. The *AD Information Helpers* (ADIH) extend an ALIA4J-based execution environment with functionality that allows to inspect the AD-related context of the running program and to interact with the execution of advanced dispatch. The communication is channeled through the standard JPDA and the AD-related data is represented by the *AD Debug Interface* (ADDI) Model. A debugger front-end for advanced-dispatching can connect to the ADDI and the JDI in order to debug the execution of advanced-dispatching and of standard Java in a program.

The front-end of our debugger is integrated into the Eclipse IDE, although any IDE with a comparable infrastructure would also be applicable. Our debugger extends the Eclipse Java debugger, which is used for the program parts that do not use advanced-dispatching, with additional user interfaces. These are Eclipse views specific to visualizing and interacting with ALIA4J's representation of advanced dispatch in order to satisfy the requirements motivated in section 2. In some cases, we have also changed the behavior of existing debugger views to adapt to AD features.

After presenting some background on ALIA4J, the JPDA and the Eclipse debugger in the following section, we will present our AD language debugger. In particular, we will present the user interface extensions in section 5. In section 6 we will shortly present the required extensions to an ALIA4J-based execution environment for communicating with the debugger. And in section 7 we will discuss our extensions to Java debugger architecture that reflect the advanced-dispatching extensions to languages and the user interface.

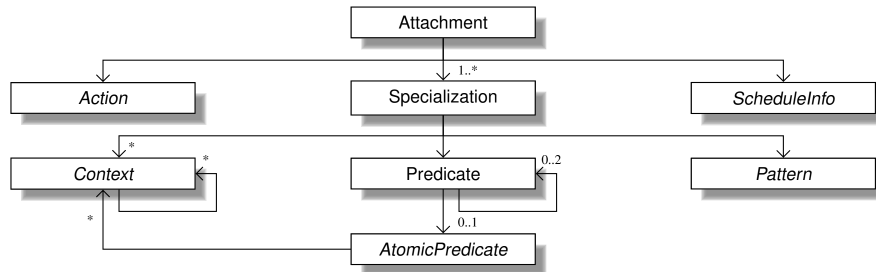



Figure 3: Entities of the Language-Independent Advanced-dispatching Meta-Model (LIAM) as UML class diagram.

## 4. Background

### 4.1. ALIA4J

ALIA4J [19] has two main components: The first component is the *Language-Independent Advanced-dispatching Meta-model* (LIAM) for expressing advanced-dispatching declarations. LIAM acts as the *format* of the intermediate representation for advanced dispatching in programs. The meta-model itself defines *categories* of concepts and how these concepts can interact, e.g., a dispatch may be ruled by *atomic predicates* which depend on values in the dynamic *context* of the dispatch. LIAM has to be *refined* with the concrete advanced-dispatching concepts of actual programming languages that are supposed to be mapped to LIAM.

Figure 3 shows the eight meta-entities of LIAM, discussed in detail in [19, Chapter 3.2], which capture the core concepts underlying the various dispatching mechanisms. The meta-entities *Action*, *AtomicPredicate* and *Context* can be refined to concrete concepts.

In short, an *attachment* corresponds to a unit of dispatch declaration, roughly corresponding to a pointcut-advice pair or to a predicate method. *Action* specifies functionality that may be executed as the result of dispatch (e.g., the body of an advice or predicate method). *Specialization* defines static and dynamic properties of state on which dispatch depends. *Pattern* specifies syntactic and lexical properties of the dispatch site; *predicate* and *atomic predicate* entities model conditions on the dynamic state a dispatch depends on. *Context* entities model access to values in the context of a dispatch, like the called object or argument values when the dispatch site is a call to a instance method. Finally, the *schedule information* models partial ordering of the action contributed to a dispatch result in relation to actions contributed by other attachments.

The actual intermediate representation of a concrete program, in turn, is a model conforming to the meta-model refinement for that language—we simplifyingly call these models *LIAM models*. Code of the program *not* using advanced dispatching mechanisms is represented in its conventional Java bytecode form.

We have validated the expressiveness of LIAM by refining it with the concrete concepts of different existing programming languages [19], namely AspectJ [8], Compose* [9], CaesarJ [10], and JAsCo [11]; refinements for MultiJava [21], JPred [6], and ConSpec [12] also exist and can be downloaded from our website[2] but are not otherwise published. For the languages AspectJ, ConSpec, and rudimentarily for Compose* we have implemented an automatic translator from source to a model according to refined LIAM.

[2] `http://www.alia4j.org`

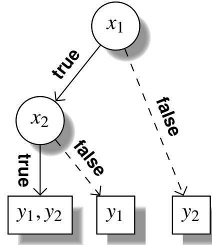



Figure 4: A dispatch function's evaluation strategy.

The second component of ALIA4J is the *Framework for Implementing Advanced-dispatching Languages* (FIAL), a framework for execution environments that allow to deploy and undeploy LIAM dispatch declarations. FIAL implements common components and work flows required to implement execution environments based on a Java Virtual Machine (JVM) for executing LIAM models. Most importantly for the purpose of this paper, it defines how to derive an execution strategy *per dispatch site* that considers all LIAM-based dispatch declarations present in the program.

The execution strategy consists of the so-called dispatch function that characterizes which actions should be executed as the result of the dispatch in a given program state. This function is represented as a binary decision diagram (BDD) [22], where the inner nodes are the atomic predicates used in the predicate definitions and the leaf nodes are labeled with the actions to be executed. For each possible result of dispatch, the BDD has one leaf node, representing an alternative result of the dispatch, i.e., which actions are to execute and in which order[3]. Figure 4 shows an example of such a BDD with the atomic predicates $x_1$ and $x_2$ and the actions $y_1$ and $y_2$. For a detailed explanation of this model, we refer the reader to [23].

Importantly for the present paper, the dispatch models and all LIAM models stay first-class during the execution of a program and can therefore easily support dynamic features of an IDE, like debugging, profiling, or testing. We have implemented several execution environments based on FIAL which apply optimizations to a different degree. All extensions to the execution environment that have been made in the course of developing the debugger presented in this paper, are made to our portable but non-optimizing execution environment called NOIRIn. It is future work, to enable similar support also in the optimizing execution environments.

### *4.2. The Java Platform Debugger Architecture*

The Java Platform Debugger Architecture (JPDA) [20] is presented in the striped box of figure 2. It defines a system consisting of 2 layers and the communication between them. From the bottom up, this are the Java Virtual Machine Tool Interface (JVMTI) layer, the Java Debug Wire Protocol (JDWP) for communication and the Java Debug Interface (JDI) layer. On top of the JDI, the debugger user interface layer is implemented. JVMTI is a native interface of the JVM. It allows to check runtime states of a program, set call-back functions for events like reaching a breakpoint, and control some environment variables. JDWP is the protocol used for the communication between the debugger—which is written in Java thus executed by a JVM—and the debuggee JVM. JDI is a mirror-based, reflective interface. The mirror mechanism maps

[3]Nested execution of, e.g., around advice in AspectJ that use the *proceed* keyword, is also supported by the execution strategy.

all data including values, types, fields, methods, events, states and resources on the debuggee JVM into mirror objects. For instance, loaded classes are mapped to *ReferenceType* mirrors, objects are mapped to *ObjectReference* mirrors, etc. Accessing fields or invoking methods of an object existing in the debuggee JVM can be performed by the debugger in the reflective way, using the mirrors.

*4.3. The Eclipse Debugger*

Eclipse provides a language-independent debug model (called the platform debug model) which defines generic debugging interfaces that are intended to be implemented and extended by language-specific implementations. The Eclipse debugger has a mirror of each relevant runtime value in the execution of the debugged program. The hierarchy of debugged artifacts is shown in figure 5. The *DebugTarget* is a debuggable execution context, in the case of Java a virtual machine, and it contains several *Thread*s which again contain *StackFrame*s. The *StackFrame* is an execution context in a suspended thread and contains *Variable*s. *Variable* has a *Value*; values can be objects with fields, which are also modeled as *Variable*. The model also defines interfaces for sending requests, like “Resume” to the debug model elements, and interfaces for handling events, like reaching a breakpoint.

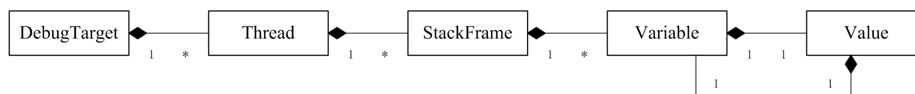



Figure 5: Eclipse platform debug model

In a typical debugging workflow the developer first locates a line in the source code where he/she assumes an error, e.g., because an exception is thrown at this line, and sets a breakpoint on that source code line. When the program is started the next time, the debugger connects to the *DebugTarget* representing its execution and sends a request to activate the breakpoint. When the execution reaches the code corresponding to the breakpoint’s line, the execution thread is suspended, and an event is sent to the debugger virtual machine, notifying it that the *Thread* is intercepted at a certain *StackFrame*. The debugger can now present this information to the developer, e.g., by highlighting the source code line that corresponds to the *StackFrame* and by presenting which *Variables* are present at the stack frame. The programmer can inspect the runtime values of these variables which requires some more interaction between the debugger and the debuggee. The developer also has different options to control the further execution of the suspended program. He/she can, for example, instruct the debugger to “Step Into” or “Step Over” the computation that is currently suspended, to “Resume” the execution or to “Terminate” the debugged process.

The Java Development Tools (JDT) of Eclipse implement the platform debug model for Java based on the JDI standard.

## 5. User-Interface for the AD debugger

The application layer presented in figure 2 consists of several views like the *Variables* and *Expressions* views which are provided by the default Java debugger in Eclipse. The AD debugger adds three views, namely the *Advanced Dispatch* view, the *Advanced-Dispatching Structure* view and the *Deployed Attachments* view. Since the local behavior in AD programs can depend on

complex dynamic context, the Advanced Dispatch view shows which context is accessed during dispatch and in which way. This is necessary for the developer to understand which actions are/are not executed at a dispatch and why this is the case.

In order to make our debugger better understood, we give a code example written in AspectJ in listing 1. For simplicity, we omit the base program with the call to *Base.foo()*. Listing 1 declares an aspect with a *before* advice which is executed when *Base.foo()* is called and the callee is an instance of class *Base*. Suppose the program is currently suspended at the point where the next instruction calls *Base.foo()*, we introduce each view in this scenario in the following paragraphs.

```
1 public aspect Aspt {
2   before() : call(* Base.foo()) && target(Base) {
3     System.out.println("advice()");
4   }
5 }
```

Listing 1: A code example of an aspect

The *Advanced Dispatch* view is the central view of the debugger showing *runtime* information about the dispatch at with the debuggee is currently suspended. It lets the developer inspect the runtime values of AD entities in the current frame, foresee the program composition flows of the next generic function invocation, directly step to any action which is going to be executed, etc. All values are presented textually in a tree viewer. The execution strategy with the dispatch function, which is represented as a BDD in ALIA4J, is additionally presented graphically. It is a special form of a branching program, for which a graphical representation should be intuitive to the developer.

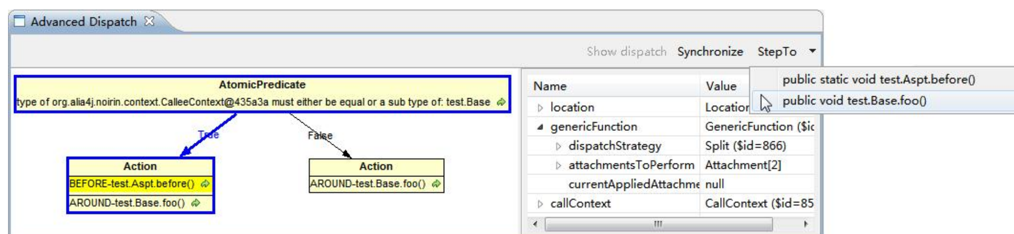



Figure 6: A snapshot of the Advanced Dispatch view showing the graphical representation for a dispatch strategy

A snapshot of the *Advanced Dispatch* view is given in figure 6. The left window frame gives a graphical representation of the execution strategy for the next dispatch which is *Base.foo()* in this example. The root node represents an *AtomicPredicate* which requires that the type of the callee should be *Base* or a subtype. Two leaf nodes show different program composition flows according to the evaluation result of the *AtomicPredicate*. The bold lines indicate the actual evaluation result and actions which are going to be performed. From this view, the developer can clearly see why and when the advice *Aspt.before()* is executed. At the top right of the view, a "StepTo" button is provided for suspension the program at any performing action. For instance, the entrance of the method *Base.foo()* can be chosen to be the next suspending point so that details of aspect activities are ignored.

Considering the complexity and importance of *Attachment*s, the Advanced Dispatch view also provides a graphical representation for *Attachments*. As shown in figure 7, the user can right click an *Attachment* in the view's tree and select "Show graph".

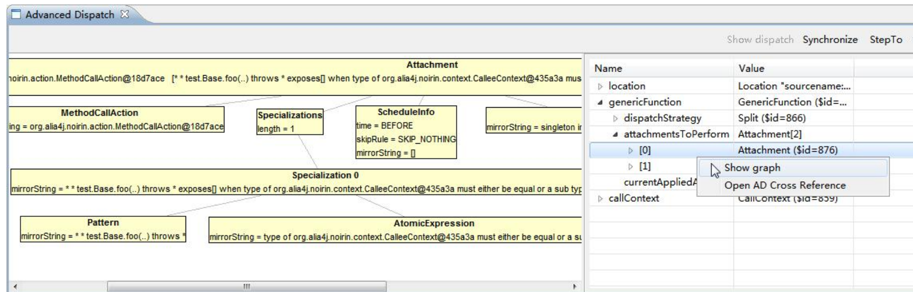



Figure 7: A snapshot of the Advanced Dispatch view showing the graphical representation for an attachment

The *Advanced-Dispatching Structure* view assists the developer to explore *static* AD information within the project scope, i.e., all dispatching declarations in the program as well as all the generic functions in the program. This view also allows the developer to navigate from a dispatching declaration to the affected generic functions and vice versa. In this example, the method *Base.foo()* may be matched by multiple *Attachment*s. However, this information can not be shown in the *Advanced Dispatch* view in an explicit way. Figure 8 shows how the *Advanced-Dispatching Structure* view explores the affection of an AD entity. In this figure, the *Attachment* matches the method *Base.foo()* which is matched by only one *Attachment*.

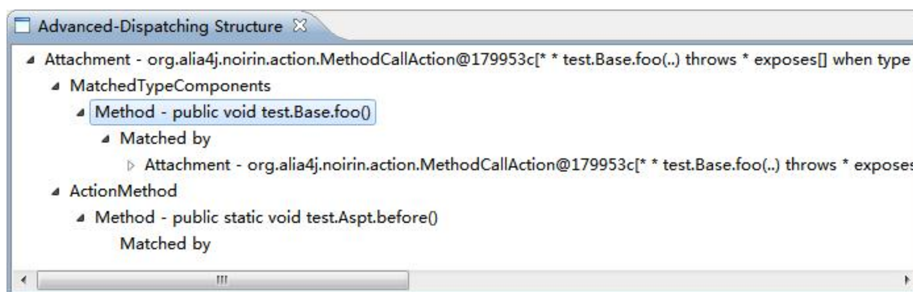



Figure 8: A snapshot of the Advanced-Dispatching Structure view

In order to dynamically deploy and undeploy attachments during runtime, the *Deployed Attachments* view is provided. It shows a textual representation of all attachments that are defined in the executing program together with a check box indicating whether the attachment is currently deployed or not. Unchecking or checking one of the items manually will lead to undeployment or deployment of the corresponding *Attachment* in the debugged program. A snapshot of the *Deployed Attachments* view is given in figure 9.

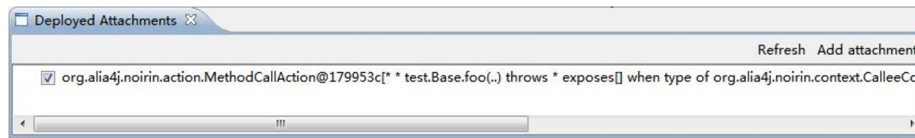



Figure 9: A snapshot of the Deployed Attachments view

## 6. Extensions to the execution environment

As mentioned in section 4.1, different implementations of execution environments exist in ALIA4J but in this paper we focus on the interpreting one called NOIRIn. The following 2 subsections describe the work flow of executing dispatch in NOIRIn. Furthermore, we will show how we have extended this work flow and added new services to enable debugging.

### *6.1. Debugging-Enabled Work Flow of Executing Advanced-Dispatching*

Figure 10 presents the work flow of deploying an advanced-dispatching declaration and of executing advanced dispatch in NOIRIn. The figure also shows the additional steps in the work flow for supporting debugging in NOIRIn. The following list describes each step in the grey box which represents the original work flow.

1. A dedicated component adapts the complied source code of an advanced-dispatching program, like an AspectJ program, to a set of attachment models conforming LIAM and sends these attachments into NOIRIn.

2. After an attachment is sent to NOIRIn, it does not take effect until it is deployed. During the process of deployment, NOIRIn extracts the pattern from the attachment in order to find out all matched dispatch sites. Then the new attachment is combined with existing dispatch logic of each dispatch site. After the preparation of the previous two steps, the actual application program starts.

3. When a dispatch site is encountered, e.g., a method call is invoked in the scope of user written code, NOIRIn intercepts this call and performs the following steps to execute the dispatch.

4. The context relating to that call, like the line number in which this call happens, the declaring class of the callee, etc., is collected by NOIRIn. According to the call context, NOIRIn finds the dispatch's execution strategy.

5. Based on the context values, the dispatch function is evaluated to decide which actions to execute.

6. For the applicable actions, an order is determined by resolving the constraints of the associated *Schedule Information*. Finally, NOIRIn executes all applicable actions in the determined order. After all actions have been executed, NOIRIn reads the next dispatch, starting over at step 3.

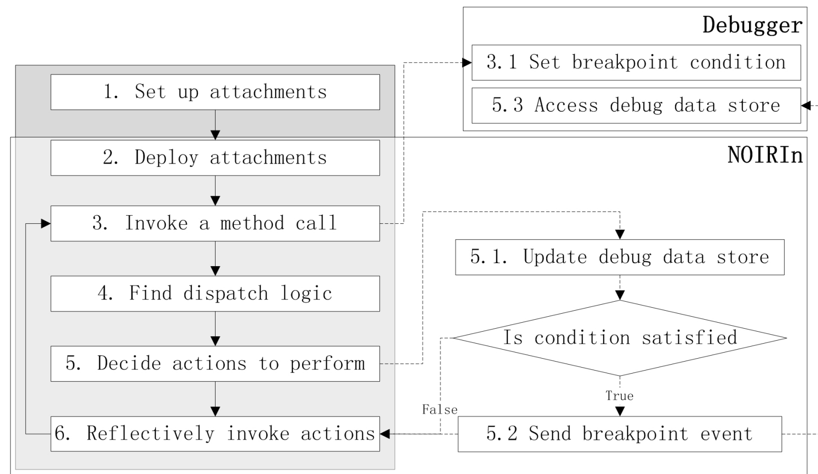



Figure 10: The work flow of intercepting a method call extended with ability supporting debugging in NOIRIn

Additional to the steps presented in figure 10 we have extend NOIRIn to support debugging. First, all needed debugging data, like the call context, the dispatch site, etc., are stored in a singleton store from which all needed data can be retrieved. Every time when the applicable actions are determined at step 5, NOIRIn writes new data into the store (5.1). Second, a conditionally executed statement is added right after the store is updated. We refer to this instruction as the *breakpoint shadow*. The debugger can set a breakpoint here to suspend the execution of NOIRIn just before the execution of the dispatch. The statement is executed conditionally and the condition is set at the time when the method is called at step 3 and the specific condition is configured according to the call context (3.1). When the program runs to the breakpoint shadow, it sends a breakpoint event to the debugger (5.2). Then, the program is suspended and the debugger accesses the store (5.3) until it asks the debuggee VM to resume.

### *6.2. Services for Enabling Debugging*

On the debugger side, the debugged entities are mirrors, code using mirrors needs to access fields and invoke methods reflectively and handle exceptions that may occur during reflection. Therefore we implemented the Advanced-Dispatching Information Helpers (ADIH) in NOIRIn such that each task can be performed just by one reflective method invocations where actually a series of methods from the standard NOIRIn API must be called. As the remote and reflective invocation of methods requires marshalling arguments and handling exceptions in a special way, the ADIH simplifies the implementation of the debugger front-end.

Besides, NOIRIn does not always provide interfaces for accessing AD declaration entities in a way suitable for debugging. Sometimes, further tasks need to be performed to adapt provided data to a required form. For instance, the debugger needs to find all matched class members according to a pattern of an attachment. NOIRIn only provides an interface for reading the pattern, the function retrieving class members was added for usage by the ADIH. Another example is that the debuggee side puts *AttachedAction* objects in the dispatch strategy. However, the dispatch strategy on the debugger side is designed to store *Attachment*. Therefore, a helper function finding the related *Attachment* according to an *AttachedAction* has been implemented.

To summarise, information helpers simplify the process of performing tasks at the debugger side and add functionalities supporting debugging which have not been available in the original NOIRIn.

### 7. Advanced-dispatching debug interface

The Advanced-dispatching Debug Interface (ADDI) extends the Java Debug Interface (JDI) by adding advanced-dispatching-related features to some existing entities and introducing new advanced-dispatching related entities. The structure of ADDI is presented in figure 11. The light grey parts are entities defined in JDI. The dark grey parts are entities defined in JDI but extended with advanced-dispatching related features. The white parts are new entities introduced in ADDI. The rest of this section describes each entity which is relevant to advanced-dispatching.

**TypeComponent** reifies a class member, i.e., a *Field* or a *Method*. A dispatch site, e.g. a field access, or a method invocation, may be matched by a pattern of an attachment. So the *TypeComponent* is extended with a function providing a list of matching attachments.

**Field** reifies a field. It extends *TypeComponent* and further distinguishes the related dispatch sites into field reading and field writing. Therefore, it is extended with functions providing a list of attachments matching its reading and writing respectively.

**ClassType** reifies a class. It is extended with a function providing a list of matching attachments of its members.

**ADMirror** reifies an advanced-dispatching related entity. It wraps an *ObjectReference* which is the mirror of an actual object in the debuggee program. Mostly, the “mirror-wrapper” *ADMirror* provides advanced-dispatching related information and functionalities by accessing fields or invoking methods of the wrapped *ObjectReference*. If the required data cannot be provided by the wrapped *ObjectReference* alone, then the request needs to use the information helpers.

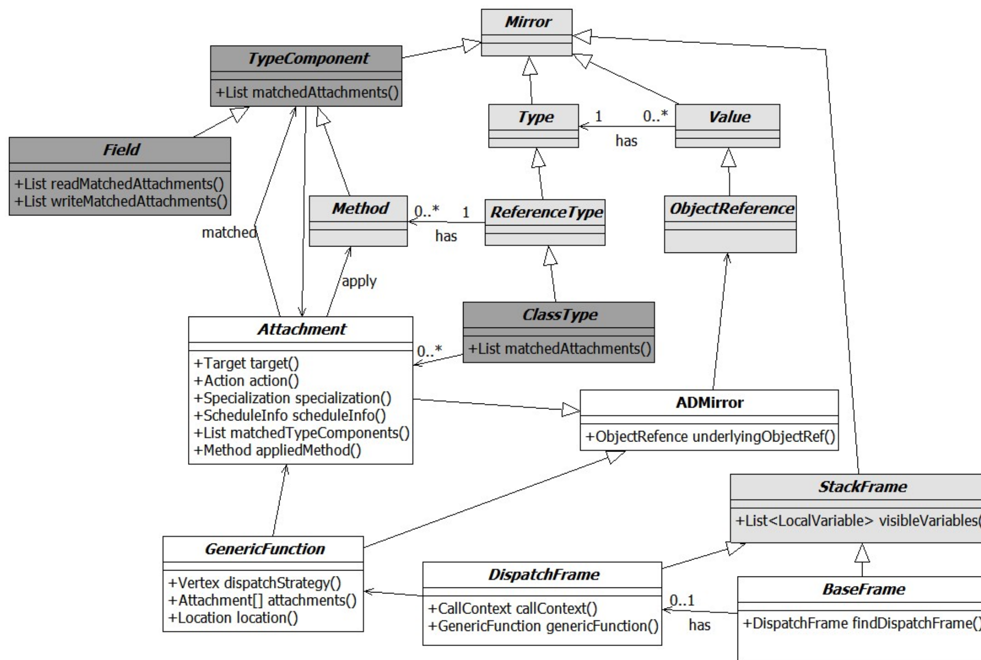



Figure 11: Simplified UML class diagram of ADDI

**Attachment** reifies an attachment. It provides interfaces for accessing components of *Attachment*. If the *Specialization* is matched, the *Action* will be invoked at the time that the *ScheduleInfo* specifies. Besides, it can provide a list of *TypeComponents* which match a pattern of the attachment and a mirror of the actual method that its *Action* implies.

**BaseFrame and DispatchFrame** reify different stack frames according to where they resides. *BaseFrame* denotes a frame which is not in the infrastructural code of ALIA4J. *DispatchFrame* is a specific frame which resides at the breakpoint shadow introduced in section 6.1 where the dispatch strategy has been computed but not yet performed. According to the execution flow in the execution environment, the *BaseFrame* is always accompanied with a *DispatchFrame*.

**GenericFunction** reifies a dispatch. The generic function is a unique identification, e.g., signature, which may be shared by multiple methods. A generic function has a list of attached actions. Besides, it has a dispatch strategy which describes which of them are executed in which order at a site performing this dispatch.

## 8. Lessons Learnt

### *8.1. Learning About the Internal Functionality of Eclipse Plug-Ins*

Though a tutorial about building an Eclipse plug-in is easy to obtain in the Internet, lacking of documentation makes realizing a desired functionality from scratch troublesome. This is also

discussed in [24]. Fortunately, there are a lot of open-source plug-ins with high code quality available which can act as examples. To implement a functionality for a plug-in, our experiences may helps.

1. First, decide which functionality is needed and think of a similar functionality provided by other plug-ins.

2. Second, you need to find the implementation of the functionality which you have located in the user interface. Eclipse provides a handy tool to support you in finding relevant code. After selecting the UI element, like view, editor, etc., press "Shift+Alt+F1" to invoke the Plug-in Spy in order to open the related source file.

*8.2. Extending Existing Plug-Ins*

We developed our debugger based on the default Java debugger in Eclipse because the functionalities for debugging base programs are the same. The core part of the Java debugger is the plug-in *org.eclipse.jdt.debug* which gives a JDI implementation. Our debugger not only *uses* the JDI implementation but also *extends* its behavior and structure. For example, when the debugger requires to disconnect, it calls *MirrorImpl.disconnectVM()* and carries out a list of disconnecting tasks. Our debugger needs to add another task into this list in order to release some resources before disconnection. Simply importing this plug-in into the library of our project does not help. Thus, we import this plug-in into a source folder.

Because the imported Java debugger plug-in code is a standard and we want to plug, unplug and maintain the extended source code easily in the future, we put all extended code in another source folders and leave the imported code intact. We modify the behavior and structure without touching the source code using AspectJ. In order to organize extended files well, they are in packages whose names are similar to the names of the packages where they are extended. For example, the imported class *StackFrameImpl* is put in the package "org.eclipse.jdi.internal". We put the extended class *BaseFrameImpl* which inherits *StackFrameImpl* in the package "extended.jdi.internal". With this naming convention, the developer can easily find out the correspondence between the imported and the extended files.

## 9. A debugging example

To demonstrate the usefulness of the presented debugger, we present a walkthrough of one debugging session. Supermarkets offer special prices for certain items and there are two types of promotion prices in this example. One type is used when an item is on sale, its price is decreased by ten percents. Another type is used when a customer has obtained enough credits from previous shoppings, he/she can get one euro bonus then. If two promotion types are applicable, the price of the item is first reduced by the constant bonus and then cut down by ten percents. Let us call it double-cut price. For example, the double-cut price for a 10-euro item is $(10 - 1) * 0.9 = 8.1$ euro.

In a supermarket system, the aspect in listing 2 is used for handling special prices. The first advice from line 4 - 6 reduces the price of an item which is on sale according to the first promotion type. The second advice from line 7 - 9 subtracts the bonus.

```
1 public aspect SpecialPrice {
2   pointcut itemGetPrice(Item i) :
```

```
3      call(* Item.getPrice()) && target(i) && !within(SpecialPrice);
4    before(Item i) : itemGetPrice(i) && if(i.isOnSale()) {
5      i.setPrice((float) (i.getPrice() * 0.9));
6    }
7    before(Item i) : itemGetPrice(i) && if(i.isBonus()) {
8      i.setPrice((float) (i.getPrice() − 1));
9    }
10 }
```

Listing 2: A code example of an aspect

Running the program in listing 3, the printed price is 8.0 euro instead of the expected 8.1 euro.

```
1 public class Main {
2   public static void main(String[] args) {
3     Item item = new Item();
4     item.setPrice(10);
5     item.setBonus();
6     item.setOnSale();
7     System.out.println(item.getPrice()); // unexpected price
8   }
9 }
```

Listing 3: Main

The following list shows the process how to use the AD debugger to find the bug.

1. Set a breakpoint at line 7 in listing 3 and launch the AD debugger.
2. The line contains multiple dispatch site. First the field System.out is read, next the method Item.getPrice is called and finally PrintStream.println is called. Thus, when the program is suspended at line 7, the developer must step over the first dispatch to suspend the JVM at the dispatch of item.getPrice().
3. Open the *Advance Dispatch* view and press the "Show dispatch" button.
4. The dispatch function is shown in the view as in figure 12. The bold lines tell the developer that three actions are going to be executed at this dispatch site. However, the order of the two advices is wrong according to the requirements. The bug is found and developer can reverse their literal order to fix this bug.

Following the classification by Eaddy et al. [18], this example introduces an *incorrect program composition* fault which occurs in the activity *dispatch function evaluation*. Only two advices from the same aspect are involved, the bug may be easily found by reading code. However, if more advices match the same call site and they are from different aspects, the conventional debugger is unable to show the program composition explicitly so that the faults become less obvious.

## 10. Related Work

Eaddy et al. [18] implemented Wicca which is a dynamic AOP system for C# applications that performs source weaving at runtime. The source code used in debugging is the woven source

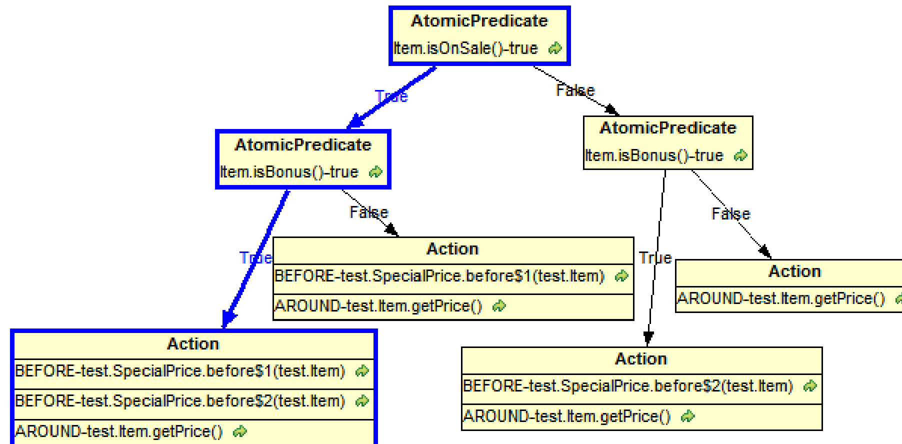



Figure 12: The dispatch function when Item.getPrice() is called in listing 3

code and only contains Object-Oriented (OO) concepts. The AD debugger is designed only for Java based languages. It uses source code written by developer as the source view and defines debugging constructs in terms of AD abstractions, such as attachment, or action.

The AspectJ Development Tools (AJDT) [13] enable the Eclipse platform to build, edit, and debug AspectJ programs. It provides a lot of features decreasing the effort in understanding and coding AspectJ programs. The *Aspect Visualiser* and *Cross References* view are most representative. The *Aspect Visualiser* is used to visualize how aspects are affecting classes in a project in a "bars and stripes" style representation. The *Cross References* view is used in the AJDT to show AspectJ crosscutting information, such as which advice a Java method is affected by advice. However, the debugger in the AJDT is problematic because the conventional Java debugger is used.

De Borger et al. [25] found the pointcut-advice models of Java-based AO-technologies, such as AspectJ, JBoss AOP, and Spring AOP to be very similar and thus defined a common debugging interface: the Aspect-Java Debugging Interface (AJDI). The AJDI aggregates JDI mirrors and the information about aspect related structure and behavior. They create events for dynamic AOP and events related to join points and advices by transforming AJDI breakpoints into JDI breakpoints. Based on the AJDI, they built the Aspect Oriented Debugging Architecture (AODA) which supports runtime visibility and traceability of aspect-oriented software systems. Compared to their work, the AD debugger is built for more general concepts which are also applicable for predicate-dispatching languages. However, locating is relatively easier in AODA because constructs defined in the architecture are language-specific.

AD-specific information provided by tools or systems for AD languages are not only provided as online debuggers as the work presented in this paper. These other approaches can be used as auxiliary approaches to understand program behavior or structure during debugging.

Pothier et al. [26] implemented an AO debugger based on an open source omniscient Java debugger called TOD [27]. The TOD records all events that occur during the execution of a program and the complete history can be inspected and queried offline after the execution. It is extended to provide *aspect mural*s that show the activity of an aspect during the execution of the

program. They also provide a view showing the execution history of the join point shadows of a particular pointcut to view which occurrences of join points matched.

The JPred [6] Eclipse plug-in provides a view showing implication relationships between predicates used for methods sharing the same signature. This is shown in terms of a Binary Decision Diagram, similar to ALIA4J's dispatch execution strategy. It indicates that a method with a more specific predicate has higher priority to be executed. Take the JPred program in listing 4 for example, the implication relationship is shown in figure 13. Compared to this view, the graphical representation of dispatch function decomposes each predicate into a set of atomic predicates and then repeated ones are removed. As shown in figure 14, it shows the evaluation order of predicates instead of the relationship between them. In contrast to our online debugger, the JPred plug-in only statically shows the decision process of dispatch.

```
1 class Test {
2   void m(i) {}
3   void m(i) when i==0 || i==1 {}
4   void m(i) when i==0 {}
5 }
```

Listing 4: A JPred program example

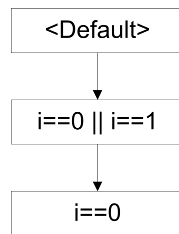



Figure 13: JPred predicate implication for method m() in listing 4

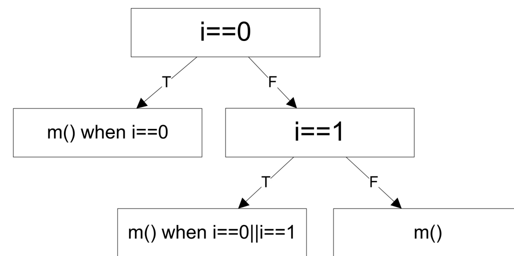



Figure 14: Graphical representation of the dispatch function when m() in listing 4 is called in the AD debugger

CaesarJ [28] is a Java based programming language, which facilitates better modularity and development of reusable components. The components are collaborations of classes, but they can modularize crosscutting features or non-functional concerns. The CaesarJ Development Tools (CJDT) extend the Eclipse's Java Development Tools (JDT) plug-in with CaesarJ-specific features. Like the AJDT, the CJDT also uses the Java debugger because CaesarJ programs can be compiled into Java bytecode. Therefore, language-specific features cannot be shown and source locations are lost in some cases.

Also for the ObjectTeams programming language an Eclipse-based IDE exists that enhances the standard JDT Java debugger [29]. The enhancement filters call frames that belong to infrastructural code and adapts the placement of breakpoints respectively the behavior of stepping through the code. While these enhancements hide the details of generated code from the developer, it still falls short in providing additional language-specifi functionality.

JAsCo [30] is an advanced AOP language tailored for the component-based field. It makes aspects reusable and provides a strong aspectual composition mechanism for managing combinations of aspects. Besides, it allows to add, change and remove aspects from the system at runtime. JAsCo Development Tools (JAsCoDT) is a tool for editing, running and debugging

JAsCo-enabled applications. JAsCoDT provides an *Introspector* which displays the connectors found within the system. It also has a *Joinpoint Lookup* view which is used for statically exploring matched join points of a hook instantiation. In the AD debugger, the *Deployed Attachment* view is similar to the *Introspector* view, attachments can be activated or deactivated by checking or unchecking. Compared to the *Joinpoint Lookup* view, the *Advanced-Dispatching Structure* view can find not only matched type components for a given attachment but also the other way around.

The approaches presented above all target specific general-purpose programming languages of the aspect-oriented or predicate-dispatching paradigms. There is another field of related work, namely work that aims to provide language-specific debugging support for domain-specific languages (DSLs). Since the ALIA4J approach also can be used to implement domain-specific language [19], we will consider this kind of related work in the future. Here, we only want to mention those related approaches that we are currently aware of.

The TIDE [31] environment is a generic debugging framework that can be instantiated for new DSLs. While it simplifies the development of a debugger for a new language, it cannot take the complete effort from the language developer. In contrast, our work provides a completely generic solution for any language that is implemented in terms of ALIA4J. Furthermore, TIDE does not specifically support advanced-dispatching features. Nevertheless, it enables a more language-specific user interface while the user interface in our work only provides visualizations of ALIA4J's abstractions.

The IDE Meta-tooling Platform (IMP) [32] is an Eclipse project aiming at providing meta-implementations of typical IDE tools. Examples are a re-usable infrastructure for syntax highlighting, refactoring support, semantic or static analyses, execution and debugging. Their focus is on providing an infrastructure for the IDE integration and the graphical user interface, but not on providing an infrastructure for the runtime part of actual debugger implementations. Nevertheless, we will consider to integrate our work with this project.

Finally, we would like to investigate the Platform-Independent Language (PIL) [33] which is an intermediate layer between DSL source code and the target platform's code. We will compare this language with ALIA4J's intermediate representation.

## 11. Conclusions and Future Work

In this paper we have presented the Advanced-dispatching Debugger Architecture (ADDA), an accordingly implemented debugger for, and our experiences of implementing it. The ADDA consists of three layers. We have implemented the application layer in terms of three new Eclipse views which offer inspection of advanced-dispatching specific runtime state, present program composition flow, explore affection of advanced-dispatching entities, deploy and undelopy advanced-dispatching features at runtime, etc. In the model layer, we have defined the Advanced-dispatching Debug Interface (ADDI) offering inspection of runtime states of advanced-dispatching entities. And for use in the data layer we adapted the ALIA4J execution environment NOIRIn to be suitable for supporting debugging.

Our future work includes developing an advance-dispatching event model, extending ADDI to language-specific debugger interfaces and supporting debugging in ALIA4J's optimizing execution environments. Furthermore, we will provide additional ALIA4J-based generic IDE tools. We will investigate using Equinox Aspects for modifying the behavior of Eclipse plug-ins.

## References


[1] C. Chambers, Object-oriented multi-methods in cecil, in: Proceedings of ECOOP, Springer Verlag, 1992.

[2] M. Ernst, C. Kaplan, C. Chambers, Predicate dispatching: A unified theory of dispatch, in: Proceedings of ECOOP, Springer Verlag, 1998.

[3] H. Masuhara, G. Kiczales, Modeling crosscutting in aspect-oriented mechanisms, in: Proceedings of ECOOP, Springer Verlag, 2003.

[4] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology, March-April 2008, ETH Zurich 7 (2008) 125–151.

[5] C. Bockisch, M. Mezini, A flexible architecture for pointcut-advice language implementations, in: Proceedings of VMIL, ACM, New York, NY, USA, 2007.

[6] T. Millstein, C. Frost, J. Ryder, A. Warth, Expressive and modular predicate dispatch for Java, ACM Transactions on Programming Languages and Systems 31 (2009).

[7] C. Clifton, T. Millstein, G. T. Leavens, C. Chambers, MultiJava: Design rationale, compiler implementation, and applications, ACM Transactions on Programming Languages and Systems 28 (2006).

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: Proceedings of the European Conference on Object-Oriented Programming, Springer Verlag, Berlin/Heidelberg, Germany, 2001, pp. 327–353.

[9] A. de Roo, M. Hendriks, W. Havinga, P. Dürr, L. Bergmans, Compose*: a language- and platform-independent aspect compiler for composition filters, in: Proceedings of WASDeTT.

[10] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, Overview of CaesarJ, in: Transactions on Aspect Oriented Software Development, volume 3880 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin/Heidelberg, Germany, 2006, pp. 135–173.

[11] D. Suvée, W. Vanderperren, V. Jonckers, JAsCo: an aspect-oriented approach tailored for component based software development, in: Proceedings of the International Conference on Aspect-Oriented Software Development, ACM Press, New York, NY, USA, 2003, pp. 21–29.

[12] I. Aktug, K. Naliuka, ConSpec: A formal language for policy specification, in: Proceedings of REM, Elsevier Science Publishers B. V., 2008.

[13] Aspectj development tools, `http://www.eclipse.org/ajdt/`, 2010.

[14] C. Bockisch, A. Sewe, Generic IDE support for dispatch-based composition, in: Proceedings of the First International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines, volume 564 of *CEUR Workshop Proceedings*, CEUR-WS, 2010.

[15] T. Lindholm, F. Yellin (Eds.), The Java Virtual Machine Specification, Addison-Wesley, 2nd edition, 1999.

[16] Information technology – Common Language Infrastructure (CLI) Partitions I to VI, ISO/IEC, Geneva, Switzerland, iso/iec 23271:2006(e) edition, 2006.

[17] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, abc : An extensible AspectJ compiler, in: Transactions on Aspect-Oriented Software Development I, number 3880 in Lecture Notes in Computer Science, Springer Verlag, Berlin/Heidelberg, Germany, 2006, pp. 293–334.

[18] M. Eaddy, A. V. Aho, W. Hu, P. McDonald, J. Burger, Debugging aspect-enabled programs (2007).

[19] C. Bockisch, An Efficient and Flexible Implementation of Aspect-Oriented Languages, Ph.D. thesis, Technische Universität Darmstadt, 2009.

[20] Java platform debugger architecture (JPDA), `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`, 2009.

[21] C. Chambers, W. Chen, Efficient multiple and predicated dispatching, in: Proceedings of OOPSLA, ACM, 1999.

[22] R. E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers C-35 (1986).

[23] A. Sewe, C. Bockisch, M. Mezini, Redundancy-free residual dispatch, in: Proceedings of FOAL, ACM, 2008.

[24] A. W. K. Nick Kirtley, P. Avgeriou, Developing a modeling tool using eclipse, in: International Workshop on Advanced Software Development Tools and Techniques, Co-located with ECOOP 2008, University of Groningen, 2008.

[25] W. De Borger, B. Lagaisse, W. Joosen, A generic and reflective debugging architecture to support runtime visibility and traceability of aspects, in: Proceedings of AOSD, ACM, 2009.

[26] G. Pothier, E. Tanter, Extending omniscient debugging to support aspect-oriented programming, in: In Proceedings of SAC, ACM, 2008.

[27] G. Pothier, É. Tanter, J. Piquer, Scalable omniscient debugging, in: Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007), ACM Press, Montreal, Canada, 2007, pp. 535–552. ACM SIGPLAN Notices , 42(10).

[28] Caesarj project, `http://caesarj.org/index.php/Caesar/HomePage`, 2010.

[29] S. Herrmann, C. Hundt, M. Mosconi, C. Pfeiffer, J. Wloka, Das object teams development tooling, Softwaretechnik-Trends 26 (2006) 42–43.
[30] Jasco, `http://ssel.vub.ac.be/jasco/index.html`, 2005.
[31] M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, J. J. Vinju, Tide: A generic debugging framework — tool demonstration —, Electron. Notes Theor. Comput. Sci. 141 (2005) 161–165.
[32] The IDE Meta-tooling Platform, `http://eclipse.org/imp/`, 2010.
[33] Z. Hemel, E. Visser, Pil: A platform independent language for retargetable dsls, in: M. van den Brand, D. Gasevic, J. Gray (Eds.), Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers, volume 5969 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 224–243.