Academic Software Development Tools and Techniques

Report on the 1st Workshop WASDeTT at ECOOP 2008

Roel Wuyts[1], Holger M. Kienle[2], Kim Mens[3], Mark van den Brand[4], and Adrian Kuhn[5]

[1] IMEC and KULeuven, Belgium
[2] Department of Computer Science, University of Victoria, Canada
[3] Département d'Ingénierie Informatique, Université catholique de Louvain, Belgium
[4] Mathematics and Computer Science, Eindhoven University of Technology, Netherlands
[5] Software Composition Group, University of Berne, Switzerland

Abstract. The objective of the 1st International Workshop on Advanced Software Development Tools and Techniques (WASDeTT-1) was to provide interested researchers with a forum to share their tool building experiences and to explore how tools can be built more effectively and efficiently. The theme for this workshop did focus on tools that target object-oriented languages and that are implemented with object-oriented languages.

This workshop report provides a brief overview of the presented tools and of the discussions that took place. The presented tools, 15 in total, covered a broad range of functionalities, among them: refactoring, modeling, behavioral specification, static and dynamic program checking, user interface composition, and program understanding. The discussion during the workshop centered around the following topics: language independent tools, tool building in an industrial context, tool building methodology, tool implementation language, and building tools with external code.

1 Introduction

In this paper we report on the 1st *Workshop on Academic*[1] *Software Development Tools and Techniques* (WASDeTT-1) that was held at ECOOP 2008. WASDeTT is planned as a workshop series that collocates with different conferences in the future; it is motivated by the observation that tools and tool building play an important role in applied computer science 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. Given the importance of tool building and the significant cost associated with it, we have initiated this workshop that allows interested researchers to share their tool building experiences and to explore how tools can be build more effectively and efficiently.

[1] In fact, the official title of the workshop was '*Advanced* Development Tools and Techniques' but during the workshop discussions it became clear that a more dedicated focus on 'academic' tools would have been a better choice.

P. Eugster (Ed.): ECOOP 2008 Workshop Reader, LNCS 5475, pp. 87–103, 2009.

The workshop series aims to address general topics such as the following questions:

- Should tool building remain a craft?
- Should research prototypes be of commercial quality?
- How to integrate and combine independently developed tools?
- What are the positive lessons learned in building tools?
- What are the (recurring) pitfalls in tool building?
- What are the good practices and techniques?
- Are there architectures and patterns for tool building?
- How to compare or benchmark tools?

Thus, the workshop series devotes particular attention to academic software development tools and tool building issues, where the term "development" is to be interpreted in the largest sense possible to encompass not only software development per se, but any subsequent evolution or maintenance activity.

The purpose of this workshop was *not* to focus on any specific kind of tool (say, refactoring or program comprehension tools), but rather to gather researchers working on a broad range of tools, with the goal of:

- providing a forum where tool builders—particularly, builders of experimental research prototypes—can talk about common issues relevant to the community of tool builders.
- providing a forum (workshop and associated journal) where researchers can present and explain *their* tool and thus not only get feedback on it but also real scientific credit.

Since this first edition was held at an object-oriented programming conference, all participants were working on academic tools that support *object-oriented* software development or tools that *analyze, manipulate or reason about object-oriented source code*.

Since the workshop has an interest in both tool-building issues and experimental tools themselves, two different kinds of contributions were solicited from potential participants:

1. either a traditional position paper with the participant's vision on tool-related issues;
2. or an actual tool submission where the participant will get the possibility of presenting his or her tool and how it was built.

We deliberately did not put any restrictions on the kinds of tools that are eligible for this workshop: the tools can be early prototypes, may have been around for years, or may have recently undergone a drastic re-implementation. Nevertheless, we do have a particular interest in experimental research tools (as opposed to commercial tools) and tools that target the object-oriented software development paradigm.

In spite of our focus on experimental research tools, we explicitly solicited position papers from software industrials as well. Not only will their participation in the workshop allow them to get a sneak preview of state-of-the-art research tools, their opinions and visions would allow builders of research prototypes to learn more about the actual needs of industry.

1.1 Origin of the Workshop and Related Workshops

The WASDeTT series builds on the experience and success of the earlier Workshop on Object-Oriented Reengineering (WOOR) series of which seven editions were co-located with ECOOP conferences. The 10th and last anniversary edition of WOOR was organized at ECOOP 2007 [1]. In spite of its specific focus on reengineering issues, WOOR traditionally gave a lot of attention to tools and tool building issues. In the last WOOR a position paper on *Must Tool Building Remain a Craft?* [2] was presented as a spring board for subsequent discussions on tool building issues in a breakout group. A result of this discussion group was that tool building issues should be further pursued within the context of a dedicated workshop series that is co-organized by the former WOOR organizers and participants, resulting in WASDeTT.

In the past, there have been other workshop that have touched on tool building issues. For example, the reverse engineering community has actively discussed tool integration in the Workshop on Standard Exchange Format[2] (WoSEF) [3] at ICSE 2000, the Dagstuhl Seminar 01041 on Interoperability of Reengineering Tools[3] in 2001, and the Workshop on Design Issues for Software Analysis and Maintenance Tools at ICSM 2005 [4]. Tool builders are increasingly leveraging external code in the form of off-the-shelf components (cf. Section 3.5). Integration issues for components is the focus of the International Workshop on Incorporating COTS-Software into Software Systems (IWICSS) that was held in 2004 at ICCBSS[4] and in 2007 at ICSE.[5]

Next to these series of workshops the journal Science of Computer Programming has devoted several special issues to academic tools and tool building. Sofar, two special issues of Experimental Software and Toolkits (EST) have been published, one in 2007 [5] and one in 2008 [6].

The observation that it is difficult to get software engineering tools adopted by industry has led to the Adoption-Centric Software Engineering (ACSE) workshop series from 2001 to 2004 [7] and more recently to the Workshop on Technology Transfer in Software Engineering [6] at ICSE 2006.

2 Accepted Papers

All submitted and accepted papers present academic research tools. These papers are available on the workshop web site at

`http://smallwiki.unibe.ch/wasdett2008/submissions/`

The tools, 15 in total, cover a broad range of topics, among them: refactoring, modeling, behavioral specification, static and dynamic program checking, user interface composition, and program comprehension. In the following, we give a brief summary

[2] `http://www.ics.uci.edu/~ses/wosef/workshop.html`
[3] `http://www.dagstuhl.de/01041`
[4] `http://www.tuisr.utulsa.edu/iwicss/`
[5] `http://www.softwareml.com/IWICSS07/`
[6] `http://web.cecs.pdx.edu/~warren/wottse/`

of the tools. In the subsequent discussion (cf. Section 3) we will refer to the tools to support our observations.

Churrasco: Supporting Collaborative Software Evolution Analysis [8].[7] This tool aims to support program comprehension of (distributed) development teams. It offers several visualizations that allow team members to explore software structures via an interactive web interface. Churrasco visualizes information obtained from FAMIX models, Subversion and Bugzilla.

The MARPLE Project: A Tool for Design Pattern Detection and Software Architecture Reconstruction [9].[8] This tool supports two reverse engineering tasks that help program comprehension, namely identification of design patterns as well as architecture reconstruction. MARPLE is realized as an Eclipse plug-in and supports systems written in Java.

Hopscotch: Towards User Interface Composition [10].[9] Hopscotch is an application framework and IDE of the Newspeak language. Newspeak is a dynamic language that is influenced by Smalltalk, Self and Beta, and implemented within Squeak. Hopscotch leverages Newspeak to realize a novel approach to interface composition.

Enforcing Structural Regularities in Software using IntensiVE [11].[10] IntensiVE is a tool suite for documenting structural source-code regularities (such as design patterns, coding conventions, etc.) in object-oriented software systems and verifying their consistency in later versions of those systems. Structural regularities are described in a declarative manner with a logic-based language.

The mCRL2 toolset [12].[11] mCRL2 is a behavioral specification language and toolset for describing communication behavior of software systems and to reason about them. The toolset has been used on a number of industrial case studies.

The Rigi Reverse Engineering Environment [13].[12] This tool supports program comprehension of software structures via a collection of fact extractors, a repository in the form of a domain-customizable exchange format, and an interactive graph visualizer. Rigi is a mature tool that is still used in research and popular in teaching, but it is currently no longer actively evolved and is in bug-fix mode.

TestQ: Exploring Structural and Maintenance [14].[13] The TestQ tool allows software developers to explore the structure and properties of the xUnit tests of their system with the goal to detect test smells. Test are visualized with hierarchical polymetric views that show metrics, so called smell flowers, and pie charts.

[7] `http://churrasco.inf.unisi.ch/`
[8] `http://essere.disco.unimib.it/reverse/Marple.html`
[9] `http://newspeaklanguage.org/`
[10] `http://www.intensive.be`
[11] `http://mcrl2.org`
[12] `http://www.rigi.csc.uvic.ca/`
[13] `http://code.google.com/p/tsmells/`

The Small Project Observatory [15].[14] The Small Project Observatory (SPO) is a visualization-based tool that supports the interactive exploration of super-repositories (i.e., repositories that host a collection of project that are developed in the context of an organization). The visualization expose relationships between projects such as developer collaborations and time-line information such a size and activity evolutions.

Developing a Modeling Tool Using Eclipse [16].[15] The Primus modeling tool supports the notion of architectural primitives (e.g., callback, layering, and push-pull) in software systems design. It is an Eclipse-based tool that offers modeling in UML with component diagrams enhanced with dedicated stereotypes and OCL constraints.

Compose: A Language and Platform Independent Aspect Compiler for Composition Filters* [17].[16] The Compose* framework is a compilation and execution platform for composition filters, which is a new language concept that can be applied to any programming language that supports the notion of message passing. Compose* currently supports Java and .NET as well as C.

The Nix Build Farm: A Declarative Approach to Continuous Integration [18]. This tool supports the idea of continuous integration or daily builds and is used for several large projects, among them the Stratego/XT program transformation toolset. Nix features a lazy functional language to describe the building and composition of packages.

Building a Refactoring Tool for Erlang [19].[17] This paper describes two major versions of a refactoring tool, called RefactorErl, for the Erlang language. The authors explain features of Erlang that make fully automatic refactoring difficult or impossible (such as dynamically constructed code), describe the supported refactorings, and distill a number of tool building guidelines based on their experiences.

Runtime Checking Java Code Using ConGu [20].[18] ConGu is an Eclipse-based tool that allows developers to write algebraic specifications that can be then be checked during the execution of the program. ConGu's compiler takes Java class files and translates the specifications to Java Modeling Language assertions.

CodeCity [21].[19] CodeCity is a tool that visualizes software interactively in 3D, following a city metaphor. The buildings within the city represent classes, which are placed within districts that represent the packages. CodeCity uses building sizes, colors and transparency to enhance comprehension.

CScout: A Refactoring Browser for C [22].[20] CScout is a refactoring tool for C that has also support for source code analysis. It allows renaming of identifiers, but also provides hyperlinked code browsing, and form-based querying and metrics calculations

[14] http://evo.inf.unisi.ch:8009/spo/go/
[15] http://www.rug.nl/informatica/onderzoek/programmas/softwareEngineering/PatternHB/tool/index
[16] http://www.ohloh.net/projects/composestar
[17] http://plc.inf.elte.hu/erlang/
[18] http://gloss.di.fc.ul.pt/congu/
[19] http://www.inf.unisi.ch/phd/wettel/codecity.html
[20] http://www.spinellis.gr/cscout/

on identifiers, functions, and files. CScout has been used on large software systems such as the Linux kernel.

The presented tools cover the whole software life cycle, ranging from system modeling (Primus) and behavioral specification (mCRL2), to frameworks for novel programming paradigms (Compose*) and user interface design (Hopscotch), to build support for a software system (Nix), to checking of programs based on static analysis (IntensiVE) and run-time execution (ConGu), to refactoring of programs (RefactorErl and CScout), and to program comprehension (Churrasco, MARPLE, Rigi, TestQ, SPO, and CodeCity).

Tools for program comprehension were especially well represented at the workshop. All of these tools offer visualizations that help to understand the structure and properties of a software system. Traditionally, program comprehension tools have extracted static dependencies of a single snapshot of a target system. This is the case for both Rigi and MARPLE. Program comprehension tools have extended functionalities in various directions. For example, they mine source code for particular code patterns or bad smells (MARPLE and TestQ), they extract information from more diverse sources such as bug-tracking systems (Churrasco), they look at multiple snapshots and track the evolution of a system over time (SPO and CodeCity), and they broaden their scope from a single system to super-repositories (SPO).

3 Workshop Structure and Outcomes

The workshop featured presentations of all 15 tools. In order to allow sufficient time for discussion, three tools were selected for longer presentations of 20 minutes each (i.e., IntensiVE, Churrasco, and Hopscotch) while the other tools were discussed in 7 minute lighting talks (a.k.a. "blitz" presentations). The longer presentations allowed room for a formal tool demo and were chosen because they seemed promising to give rise to interesting interactions and discussions.

Presenters were instructed that they should introduce their tool in a nutshell (i.e., its purpose, its strength and its main weaknesses) and to then focus on tool builder issues (i.e., lessons learned that are of interest for fellow tool builders). Each talk was followed by a short round of discussion that allowed for a few questions.

After the talks a short plenary discussion took place in order to plan the interactive part of the workshop. The following topics were proposed by organizers and participants:

(i) language independent tools: How can we build tools that work across multiple languages?
(ii) tool building in an industrial context: How to build tools that get accepted in industry?
(iii) data interoperability among tools: How to exchange data between tools and how to process this data?
(iv) maturation of tools: How to grow a tool from an early prototype into a mature tool or framework?
(v) tool building methodology: How—and to what degree—can we adopt established software engineering techniques for building research tools?

(vi) tool building in teams: How to build tools in larger—and possibly distributed—teams?

(vii) tool implementation language: How does the choice of a programming language impact the building of a tool, its usability, and the context in which the tool can be applied?

It was decided to pick the four most popular topics from the above list by vote, and to allocate about 25 minutes for discussion for each topic. Each of the organizers did introduce and moderate one topic. The selected topics were (i), (ii), (v) and (vii) and the discussions are summarized in Sections 3.1–3.4, respectively.

By polling the audience we found that nearly everybody was actively engaged in tool building or had concrete tool building experiences. This high level of expertise was reflected by engaging and lively discussions.

During the decision on the topics a discussion ensued about the observation that a significant number of tools depend on external code in the form of other tools, libraries, or frameworks. This discussion is summarized and expanded upon in Section 3.5.

3.1 Language Independent Tools

Language independence is an important feature for a tool. It is equally important for tool users as well as tool developers. From a user perspective, a tool that can be applied to a portfolio of languages can be an important incentive because once the general principles of a tool are understood, it can be applied to a larger variety of different software systems. From a developer perspective, deciding on the target language(s) is an important research decision that also can have a significant impact on the design and architecture of the tool.

To better understand the issue of language independence, it is instructive to look at the history of reverse engineering tools. Many reverse engineering tools, especially the ones developed in the late 80s and early 90s, supported only a single programming language (e.g., MasterScope for Lisp, FAST for Fortran, and Cscope for C [23]). Since these tools consisted of a single front end, there was often a tight coupling between the targeted language and the rest of the system [24]. This rather tight coupling was often not intentional and thus not realized by the tool builders. Adding of a new front end for a different language, which should have been a conceptually simple task, proved in practice to be quite difficult or infeasible. This observation led to the proposal of a clean separation between extraction and analysis by means of a language-independent, general representation that captures the semantics of multiple source languages [24].[21]

Language independent representations enable a decoupling of the processing for different source languages from subsequent analyses. The vision here is that analyses can operate across all of the source languages without modification. But can we come up with a (intermediate) language that we can use to represent and reason about all languages? One approach to tackle this question is based on the observation that it depends

[21] Note the similarity to the domain of compiler construction. For instance, UNCOL was an attempt to define a unified, executable intermediate representation for diverse programming languages.

on the purpose of the language and the tools that leverage the language. Important considerations are for instance:

- semantic rigor of the language elements
- granularity of the language elements
- abstractions that generalize over elements of different languages
- characteristics and diversity of the targeted languages

The more semantic precision we demand from a language, the more difficult it becomes to define and describe such a language. Semantic rigor is needed if the tool has to support sound analyses (such as fully automated code transformations). In this case, the language needs to be fine grained (i.e., allowing to express information at the level of abstract syntax trees and control flow graphs). Examples of such tools are IntensiVE, ConGu, and CScout. In contrast, there are tools that do neither require sound analyses nor fine-grained information. This is the case for program understanding and reverse engineering tools that provide abstractions over the target language such as Rigi, TestQ, and CodeCity. For such kinds of tools language independent representations seem
feasible.

The FAMIX meta-model of the Moose tool provides a language-independent representation of object-oriented features, the core model, which can be extended with language-specific information via subclassing. FAMIX's core model consists of entities to represent classes, methods, attributes, method invocations, field accesses, and inheritance relationships. A weak approach to language independence is provided by Rigi's exchange format, called RSF. RSF enables to define a meta-model via attaching types in the form of labels to nodes and arcs. Multiple meta-models can be shared by following naming conventions on the type names. However, both FAMIX and RSF are so-called "middle-level" representations; they have not been designed for the representation of fine-grained information and as a consequence are not suitable for this purpose.

To achieve language independence it is necessary to find suitable abstractions that generalize over multiple languages. In this respect, the challenge for a suitable language independent representation is to find appropriate abstractions that preserve the needed semantic rigor over multiple languages. The IntensiVE tool for example provides abstractions in the form of intentions over properties of the source code. Another example of abstraction are intermediate representations such as Java bytecode and .NET CIL that are targeted by a number of diverse languages.

However, finding suitable abstractions can be difficult or may turn out to be infeasible. This leads to the observation that language independence should not be pursued at all cost. From a research perspective, language independence is often not a crucial requirement. For example, validation of a novel research idea with a tool prototype can focus typically on a single language. From an economic perspective, it may be more cost effective to re-implement functionality for a new source language instead of expending the effort of first redesigning the system for language independence to then benefit from the reuse of the language independent code. An approach to simplify retargeting for a new language is model-driven development. The authors of the RefactorErl tool follow this approach and describes it as follows: "This is a declarative approach which maintains the refactoring-specific lexical, syntactical and static semantical rules

of the investigated language as data. Modifying these data should result in the (as far as possible) automatic adaptation of the code of all the components of the refactoring tool" [19].

It seems important that researchers clarify the term language independent because it has different meanings. For example, language independence could mean that the tool is designed in such a manner that it is feasible to support a new language. Depending on the language, this may require more or less effort. In practice, this often means that it is only feasible to add a language that has certain characteristics; for instance, the language may have to support a certain paradigm such as object-oriented or procedural as opposed to functional, or it may have to be statically typed as opposed to dynamically typed. Furthermore, language independence could also mean that the tool works on any or most kinds of languages because it abstracts away from language-specific features. This approach is pursued by lexically-based clone detection tool such as CCFinder. Last, language independence could also mean that the tool targets systems that are composed of multiple (programming) languages. One approach to effectively support multiple languages is to integrate them into a common meta-model (e.g., GUPRO [25] and the ASF+SDF Meta-Environment [26]).

From a practical engineering point of view, it may be better to focus on implementing full support for one language first, and then to add support for multiple languages later on. However, language independence should be taken into account as an important requirement from the start when designing and architecting the tool.

3.2 Tool Building in an Industrial Context

Transforming academic tools into commercial tools involves quite some effort. The best way is to do this outside the academic environment, in a so-called spin-off company. There are quite a number of success stories but also quite a number of failures. The Software Improvement Group [22] is an example of a very successful spin-off company. This company made a commercial product of the DocGen tooling [27] and applied this tooling to quite a number of large-scale projects. The original idea of selling the DocGen tooling was transformed into selling consultancy based on the DocGen tooling. The advantage of this way of working is that the requirements on the DocGen tooling were less severe since the users are always (in-house) experts.

RefactorErl [19] is an example of a tool that was built for an industrial partner (in this case Ericsson) on demand. This a very fortunate situation which is established via a long lasting cooperation between this company and the university. Such an opportunity enables a quick transformation of research ideas into a commercially relevant tooling. However, the risk is that the university is transformed into a company. Furthermore, it puts some responsibility on the developers with respect to maintenance of the tooling.

The application of academic tools in an industrial context is in general not trivial. There are two main reasons for this phenomena. The most important one is the level of maturity of the academic tools. In most cases the academic tools are prototypes and application on industrial scale usually involves quite some effort. In some cases it means a complete reimplementation of the tooling. Second, the industry wants guarantees about support, maintenance, documentation, training, etc. Commercial tools and tool vendors

[22] http://www.sig.nl

can and will offer this support whereas in an academic setting quite often one is depending on a small-scale development team.

In a recent large-scale research project, "Ideals," [23] one of the goals was to use industry-as-laboratory. The tools and methodology developed in this research project had to be deployed at the industrial partner ASML [24], a company which builds and sells high-tech products (waffersteppers). They have a huge code base and the scope of this research project was to improve the maintainability of the code base. Besides performing research the goal of the project was to transition the developed technology as fast as possible to the company. The encountered problems were in the area of finding managers who wanted to participate in applying this new technology, furthermore the maturity of tooling, development of documentation and development of teaching material were considered as problematic. Eventually, one or two research results were transferred to ASML.

In the middle of the 90's ASF+SDF [26] was used to prototype a domain specific language for describing financial products [28]. This work was done in cooperation with a Dutch bank and a large software company. The specifications and generated C-code was transferred to both the bank and software company. This software company sold this product to another Dutch bank. Around 2005 a small problem in the software was detected and this triggered some maintenance on the specification. Fortunately, we had still the original specifications because the software company had thrown away the specifications and only kept the generated C-files. The reason for this was that they did not understand the specifications and thought they were not important.

These examples show that transferring academic tooling to industry is a huge effort that should not be underestimated. The success of the transfer depends on many factors but the most important ones are commitment of the company and commitment of the researchers involved.

3.3 Tool Building Methodology

The developers of research tools in computer science are quite often also involved in teaching courses on programming and software engineering. The traditional software engineering courses address topics like, requirements engineering, design, architecture, coding (standards), testing, software process, etc. Do we apply (some of) these techniques when developing our academic software? Only a few participants indicate that they have used, or should use some of the established software engineering techniques. Of course, the underlying hypothesis is that the quality of the software and the efficiency of development increases when applying software engineering techniques, which leaves more time for writing (scientific) papers.

There are examples of dedicated processes that have been proposed for industrial researching and academic tool building. Extreme Researching (XR) is a process specifically designed for applied research and development in a distributed environment that has to cope with changing human resources and rapid prototyping [29]. It is an adaptation of Extreme Programming (XP) and has been developed by Ericsson Applied

[23] http://www.esi.nl/Ideals
[24] http://www.asml.com

Research Laboratories to support distributed telecommunications research. XR is based on several core principles taken from XP (short development cycles, test-driven development, collective ownership, and discipline) and encodes them in a set of activities (e.g., remote pair programming, unit testing, collective knowledge, coding standards, frequent integration, and metaphor). Dedicated tool support is available for XR with a web-based portal. Notably, the authors of XR estimate, based on three projects, that their process has yielded an increase of output of around 24% and reduced project-overrun time on average by half. Kienle and Müller explore the current state of tool building practices in academia with the aim to improve upon the state-of-the-art [30]. Based on a literature survey, they propose a number of desirable characteristics for a process in academia. Such a process should be feedback-based (soliciting of input from users), iterative (as opposed to waterfall), prototype-based, lightweight (minimizing of artifacts), and adaptive (to accommodate changing project requirements). Based on the identified requirements, they introduce a tailorable process framework based on work products. The work products address issues such as tool requirements (both functional and non-functional), prototyping (technical and user interface prototype), and tool architecture. A tailorable framework is needed because "the individual [tool] projects seem too diverse to be accommodated by a single, one-size-fits-all process" [30].

It appears that modern software engineering techniques, such as agile development principles, are more frequently applied by researchers when building their tooling. A considerable number of participants indicate that they use (unit) testing and everybody uses version management systems. One of the reasons not to use traditional software engineering techniques is related to the domain: you do not know what you will need to do, so it is hard to write a requirements document up-front. So you start with a vague idea, a prototype, and you extend it. So agile development works better in a research context: you discover more of the exact problem domain as you go with your research. In a research context even a full-fledged agile development methodology (let alone a heavier process) can even be too structured specially when developing prototypes to support (scientific) papers, however as soon as a tool becomes increasingly popular and is used by other researchers, a more structured approach to software development is needed.

In this context, the following question is of interest: Do universities rather produce people that are good developers or good researchers? In order to perform good research—especially in the domain of software engineering—it may be necessary to develop good software. Furthermore, is the difference between industry and academia that big? Quite often industry applies agile software development even when the official policy is a heavy software process. The freedom in academic software development is also relative. If your tool becomes a success, the freedom in development is gone. In this case, you have pressure to provide architectural descriptions and good documentation, and you even may have to freeze interfaces or features. User want to have stable tooling and fellow developers want to have a stable architecture. But it is also important to have well-defined and stable interfaces, so that other developers and researcher can develop their own components. It is not bad to start with a small quick-and-dirty prototype for a paper, but when do you decide that it makes sense to set up Bugzilla, write architectural documentation, introduce coding standards, etc. If you start introducing more formality into your tool

development too early, it may end up being a waste of time and unnecessarily bogging down the rapid prototyping of tool functionality; if you introduce it too late you may waste time and resources as well because the tool architecture, interfaces and the code have to go through a major restructuring in order to stabilize further development.

The overall conclusion is that application of rigorous software engineering principles is not (always) possible, but some agreement on principles, architecture, and tooling can safe a lot of time and energy. So, everybody needs to use the techniques that fits for them best—as long as they are made explicit.

3.4 Tool Implementation Language

Perhaps most importantly, never forget why you are building your prototype. If it is built for the sole purpose to validate a research idea then you can create your tool as a prototype in whatever language that allows you to be most productive so that you can focus on your research contribution and not your tool. For example, the implementation of SOUL was done in Smalltalk, which was an appropriate approach because the goal was to demonstrate how a logic language can be integrated with an object-oriented language, and how the logic language can then be used as a meta-programming language for that object-oriented language. In this context, there was no benefit in implementing SOUL in a more efficient language such as C++ or more popular language such as Java. Note also that if industry decides to pick up research ideas embodied in your prototype, regardless of the language it is implemented in, it will be rewritten anyway.

Certain language features can be quite helpful for tool design. For example, C++ generics in combination with the Standard Template Library (STL) can simplify the experimentation with different strategies or algorithms. For example, the authors of mCRL2 say that "primary motivations for the choice of C++ were advanced language facilities for creating library interfaces, the availability of the C++ Standard Library, and facilities for generic programming. Generic programming not only reduces the duplication of code, it also makes it easier to adapt algorithms" [12]. On the downside, the author of CScout reports that the use of STL complicates debugging with gdb because "gdb provides a view of the data structures' implementation details, but not their high-level operations" [22].

Another ingredient when selecting the implementation language can be the principle of "eating your own dog food." The ASF+SDF Meta-Environment [26] is an environment for developing language descriptions, both syntax as well as semantics. A number of components (e.g., parse table generator, compiler, and well-formedness checker of SDF definitions) in this system are developed using ASF+SDF itself. This approach enables the immediate checking of functionality and expressiveness of the underlying formalism.

The choice of the implementation language is of course also influenced by education. If the students are mostly or exclusively introduced to Java in their courses, it is likely that when they become researchers they will develop their software in Java as well. This can be a drawback. The first version of the ASF+SDF Meta-Environment was implemented in Lisp, a very popular language in the 80's for prototyping. Researchers that joint the project later on were not so familiar with Lisp and this caused quite some problems when doing maintenance.

Table 1. The presented tools and the components that they are leveraging

Tool	Leveraged Components
Churrasco	FAMIX (meta-model), MOOSE (fact extraction), GLORP (repository), SVG (visualization)
MARPLE	Eclipse: JDT (Java fact extraction), GEF (visualization), Glassfish (distributed computing), Weka (clustering)
Hopscotch	Smalltalk
IntensiVE	Eclipse JDT (Java fact extraction), javaconnect (interoperability from Smalltalk to Java), SOUL (querying), Mondrian (visualization), Star Browser (user interface)
mCRL2	ATerms (repository), Boost (utility), C++ Standard Library (utility)
Rigi	Yacc (C fact extractor), Tk (user interface), GraphEd (graph layout)
TestQ	Fetch toolchain: Source Navigator (fact extraction), CDIF2RSF (format transformation), pmccabe (C/C++ metrics), JavaNSCC (Java metrics), Crocopat (querying), Guess (visualization)
SPO	MOOSE (fact extraction), Seaside (user interface), SVG (visualization)
Primus	Eclipse: OCL (modeling support), UML2 (user interface), UML2Tools (visualization)
Nix Build Farm	ATerms (term rewriting)
RefactorErl	XML (configuration), Emacs (user interface)
ConGu	Eclipse (user interface)
CodeCity	FAMIX (meta-model), MOOSE (Smalltalk fact extraction), iPlasma (Java/C++ fact extraction), MSE exchange format (repository), Jun/OpenGL (visualization)
CScout	BtYacc (C fact extractor), STL (utility), mySQL (repository and querying), dot (visualization)

3.5 Building Tools with External Code

It seems that researchers are increasingly leveraging components to assemble their tools instead of building them from scratch. In fact, this is reflected by most of the presented tools. Table 1 shows the workshop's tools along with examples of the components that they are leveraging in order to realize the tools' functionalities. We use the term *components* to denote external code that is packaged in such a way that reuse is facilitated. In this context, a suitable working definition of a component is given by Meyer, who characterizes it as "an element of software that can be used by many different applications" [31, page 1200]. Leveraging components for tool building has been dubbed component-based tool development (CBTD) [32].

There are many examples of suitable components that can be used for tool building. Drawing from the components in Table 1, we can classify components according to standard functionalities as follows:

repository: Tools typically need to store information in some form or another. This can be accomplished with exchange formats such as Moose's MSE[33] (CodeCity), ATerms [34] (mCRL2), the Rigi Standard Format (RSF), CDIF (TestQ), or XML (MARPLE); alternatively, tools can leverage relational databases (CScout uses mySQL and PostgreSQL) or object-relational database mappers (Churrasco uses

GLORP[25]). An advantage of using repository components is that they often have some form of modeling support and also come with a query and/or transformation language. For example, support for modeling of information about source code is provided by FAMIX[26] and Rigi.

fact extraction: Tools for software engineering have to extract information about the target system. Almost all presented tools extract static information from source code (as opposed to dynamic information collected during program execution). Since the building of extractors for complex languages such as C++ is a time-consuming and error-prone task, many tools rely on external fact extractors. For example, TestQ relies on SourceNavigator. CodeCity uses MOOSE for Smalltalk and iPlasma[27] for C++ and Java code. MARPLE and IntensiVE query the parse tree of the Eclipse JDT. One can also use parser generators to simply the construction of a custom extractor. Rigi has a parser for C that is based on Yacc (no longer supported), and CScout's parser is based on BtYacc.[28]

user interface and visualization: Tools have to provide some form of visualization and user interface to show and manipulate information. Several tools offer user interfaces based on web technology. CScout has a simple HTML-based interface while Churrasco and SPO have more sophisticated Web 2.0 like interfaces. Both Churrasco and SPO use Scalable Vector Graphics (SVG) with JavaScript for interactive visualizations. SPO also uses the Seaside[29] web application framework. Another popular strategy is to integrated the tool with an IDE. Several tools are realized as Eclipse plug-ins (MARPLE, Primus, and ConGu), while RefactorErl leverages Emacs. To visualize software structures several tools use interactive graph editors. IntensiVE is based on the Mondrian[30] visualization engine, TestQ is based on GUESS,[31] and MARPLE uses the Eclipse Graphical Editing Framework (GEF). In contrast, CScout provides static graphs based on dot.

data structures and algorithms: General utility libraries for data structures and algorithms can reduce implementation effort for tools. Both mCRL2 and CScout are implemented in C++ and use its Standard Library extensively.

An interesting question that should be further pursued is to what extent CBTD differs from implementing tools from scratch, and what are the potential benefits and drawbacks of both development approaches? The developers of the Primus tool report several lessons learned of building their tool on top of Eclipse [16]. For example, plugin-in develop can be challenging because of lack of in-depth documentation and code examples. Available plug-ins can vary considerably in their maturity and immature plug-ins may evolve rapidly and have unstable APIs. Also, APIs of plug-ins may not be powerful or flexible enough. For example, the OCL editor plug-in does not provide sufficient

[25] `http://www.glorp.org`
[26] `http://www.iam.unibe.ch/~famoos/FAMIX/`
[27] `http://loose.upt.ro/iplasma/`
[28] `http://www.siber.com/btyacc/`
[29] `http://www.seaside.st/`
[30] `http://moose.unibe.ch/tools/mondrian`
[31] `http://graphexploration.cond.org/`

detail about query errors for Primus' purposes. On the positive side, the Eclipse online community is active and responsive.

4 Conclusion and Outlook

The first WASDeTT turned out to be a stimulating event with 15 presentations that covered a variety of software engineering tools, formal tool demos as part of several presentations, and thought-provoking discussions among the participants. The discussed topics—language independent tools, tool building in an industrial context, tool building methodology, tool implementation language, and tool building with external code—showed that tool building issues are of interest to many researchers. This workshop generated first promising results, but it seems worthwhile to further pursue these discussion topics in subsequent workshops. Indeed, the 2nd WASDeTT[32] was already held (collocated with ICSM 2008), which had a special focus on tool building in an industrial context with four invited talks.

We already made the point that one important goal of this workshop series is to enable researchers to publish about their tools so that they can get scientific credit for their tool building efforts. To further this goal, a selection of the best tool submissions will be published in a special issue on *Experimental Software Toolkits* (EST) of Elsevier's *Science of Computer Programming* journal [5]. Two organizers of this workshop (Mark van den Brand and Kim Mens) will act as the editors of this EST issue.

Acknowledgments

Many thanks to the workshop presenters and participants. We also gratefully acknowledge the dedicated work of the program committee.

References

1. Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H. (eds.): Proceedings of the ECOOP 2007 Workshop on Object-Oriented Re-engineering (WOOR 2007) – 10th anniversary edition (2007), http://smallwiki.unibe.ch/woor2007/
2. Kienle, H.M.: Must tool building remain a craft? In: Demeyer, S., Guéhéneuc, Y.G., Mens, K., Wuyts, R., Ducasse, S., Gall, H. (eds.) Proceedings of the ECOOP 2007 Workshop on Object-Oriented Re-engineering (WOOR 2007) – 10th anniversary edition (2007)
3. Sim, S.E., Koschke, R.: WoSEF: Workshop on standard exchange format. IEEE Software Engineering Notes 26(1), 44–49 (2001)
4. Jin, D.: Design issues for software analysis and maintenance tools. In: IEEE International Workshop on Software Technology and Engineering Practice (STEP 2005), pp. 115–117 (2005)
5. van den Brand, M.: Guest editor's introduction: Experimental software and toolkits (EST). Science of Computer Programming 69(1–3), 1–2 (2007)
6. van den Brand, M.: Guest editor's introduction: Second issue of experimental software and toolkits (EST). Science of Computer Programming 71(1–2), 1–2 (2008)

[32] http://wasdett2.wikispaces.com/

7. Balzer, B., Litoiu, M., Müller, H., Smith, D., Storey, M., Tilley, S., Wong, K.: 4th International Workshop on Adoption-Centric Software Engineering (ACSE 2004), pp. 1–2 (2004)
8. D'Ambros, M., Lanza, M.: Churrasco: Supporting collaborative software evolution analysis. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
9. Arcelli, F., Tosi, C., Zanoni, M., Maggioni, S.: The MARPLE project: A tool for design pattern detection and software architecture reconstruction. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
10. Boykov, V.: Hopscotch: Towards user interface composition. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
11. Brichau, J., Kellens, A., Castro, S., D'Hondt, T.: Enforcing structural regularities in software using IntensiVE. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
12. Groote, J.F., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y., van Weerdenburg, M., Wesselink, W., Willemse, T., van der Wulp, J.: The mCRL2 toolset. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
13. Kienle, H.M., Müller, H.A.: The Rigi reverse engineering environment. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
14. Breugelmans, M., Rompaey, B.V.: TestQ: Exploring structural and maintenance characteristics of unit test suites. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
15. Lungu, M., Lanza, M.: The small project observatory. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
16. Kamal, A.W., Kirtley, N., Avgeriou, P.: Developing a modeling tool using Eclipse. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
17. de Roo, A., Hendriks, M., Havinga, W., Durr, P., Bergmans, L.: Compose*: A language and platform independent aspect compiler for composition filters. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
18. Dolstra, E., Visser, E.: The Nix Build Farm: A declarative approach to continuous integration. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
19. Horváth, Z., Lovei, L., Kozsik, T., Kitlei, R.: Building a refactoring tool for Erlang. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
20. Vasconcelos, V.T., Nunes, I., Lopes, A., Ramiro, N., Crispim, P.: Runtime checking Java code using ConGu. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
21. Wettel, R., Lanza, M.: CodeCity. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)

22. Spinellis, D.: CScout: A refactoring browser for C. In: Mens, K., van den Brand, M., Kuhn, A., Kienle, H.M., Wuyts, R. (eds.) 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1) (2008)
23. Chen, Y., Nishimoto, M.Y., Ramamoorthy, C.V.: The C information abstraction system. IEEE Transactions on Software Engineering 16(3), 325–334 (1990)
24. Reubenstein, H., Piazza, R., Roberts, S.: Separating parsing and analysis in reverse engineering. In: 1st IEEE Working Conference on Reverse Engineering (WCRE 1993), pp. 117–125 (1993)
25. Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program comprehension in multi-language systems. In: 5th IEEE Working Conference on Reverse Engineering (WCRE 1998), pp. 135–143 (1998)
26. van den Brand, M., Bruntink, M., Economopoulos, G., de Jong, H., Klint, P., Kooiker, T., van der Storm, T., Vinju, J.: Using The Meta-environment for Maintenance and Renovation. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007), pp. 331–332. IEEE Computer Society Press, Los Alamitos (2007)
27. Deursen, A., Kuipers, T.: Building documentation generators. In: Proceedings International Conference on Software Maintenance, pp. 40–49. IEEE Computer Society, Los Alamitos (1999)
28. van den Brand, M., van Deursen, A., Klint, P., Klusener, S., van den Meulen, E.: Industrial applications of ASF+SDF. In: Wirsing, M., Nivat, M. (eds.) AMAST 1996. LNCS, vol. 1101. Springer, Heidelberg (1996)
29. Chirouze, O., Cleary, D., Mitchell, G.G.: A software methodology for applied research: eXtreme Researching. Software—Practice and Experience 35(15), 1441–1454 (2005)
30. Kienle, H.M., Müller, H.A.: Towards a process for developing maintenance tools in academia. In: 15th IEEE Working Conference on Reverse Engineering (WCRE 2008), pp. 237–246 (2008)
31. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
32. Kienle, H.M.: Component-based tool development. In: Frontiers of Software Maintenance (FoSM) at ICSM 2008 (2008)
33. Kuhn, A., Verwaest, T.: FAME, a polyglot library for metamodeling at runtime. In: Workshop on Models at Runtime, n. 10 (2008)
34. van den Brand, M., de Jong, H., Klint, P., Olivier, P.: Efficient Annotated Terms. Software, Practice & Experience 30, 259–291 (2000)