The Soot framework for Java program analysis: a retrospective

Patrick Lam*, Eric Bodden†, Ondřej Lhoták* and Laurie Hendren§
* University of Waterloo
† Technische Universität Darmstadt
§ McGill University

Abstract—**Soot is a successful framework for experimenting with compiler and software engineering techniques for Java programs. Researchers from around the world have implemented a wide range of research tools which build on Soot, and Soot has been widely used by students for both courses and thesis research. In this paper, we describe relevant features of Soot, summarize its development process, and discuss useful features for future program analysis frameworks.**

I. INTRODUCTION

The Soot framework for Java program analysis was first released in 2000. Soot enables its users to develop static analysis tools for Java programs. Researchers have implemented a wide range of research tools[1] which build on Soot, describing their work in numerous international conference and workshop publications. In particular, Soot has been widely used by students for both courses and thesis research. This paper summarizes our experience developing the Soot framework, describes its key features, and contributes our thoughts about program analysis frameworks in general.

At its core, Soot is a compiler; it accepts Java Virtual Machine bytecode or Java source code, and it (mainly) emits Java bytecode. Figure 1 depicts the main parts of the Soot workflow. Researchers extend Soot by implementing additional compiler passes which analyze or transform Soot intermediate representations. The key features of Soot include a simplified three-address intermediate representation of Java bytecode; a number of pointer analysis and call graph construction algorithms; and the ability to produce executable Java bytecode as output.

Soot has been used for a broad range of applications, including numerous program transformation tools, research in pointer analysis, analysis of concurrent programs, symbolic execution, and the static analysis portion of hybrid static and dynamic analysis approaches.

The remainder of this paper is structured as follows. We begin by summarizing Soot's features in Section II, and continue by explaining how Soot users can hook their analyses into Soot in Section III. We continue with a discussion of the Soot development process and community in Section IV; in particular, Section V describes notable changes that we made to Soot throughout its evolution, and outlines specific features that we would like to see in future versions of Soot or other program analysis frameworks. Finally, we conclude with some reflections on Soot and reasons for its success, as well as suggestions for future program analysis frameworks in Section VI.

II. SOOT FEATURES

We continue by discussing relevant intraprocedural and interprocedural features of the Soot framework, as well as Soot's provisions for outputting analysis and transformation results. We also mention some applications of these features.

A. *Intraprocedural Features*

Soot's fundamental intermediate representation is Jimple, a typed three-address code. The creation of Jimple was motivated by the difficulty of directly analyzing Java bytecode: although it is possible to construct a control-flow graph for Java bytecode, the implicit stack masks the flow of data and thus makes the bytecode quite difficult to analyze. In particular, at a given bytecode instruction s, it is not at all obvious which previous s' produced the stack-based inputs of s. Storing data in named local variables, rather than on the implicit stack, makes the local flow of data (along Jimple's Control-Flow Graph) much more obvious. The local variables in Jimple are split according to definition-use chains. Soot also has other intermediate representations: Shimple [Uma06] is an SSA-based version of Jimple; Baf and Grimple are used to output bytecode; and Dava is an abstract syntax tree-based intermediate representation produced via decompilation of the Jimple IR.

Many Soot users would like to connect the results of their analysis to the original Java source code. Because Java bytecode includes the class and method structure of the original source code, Soot analyses can always leverage class and method name information. Soot can also provide line number and variable name information for the methods it is analyzing. In particular, when Soot is executed with the appropriate command-line flag, and if the source code was compiled with debug information, Soot will attach line number information to relevant Jimple statements. Soot can also make original variable names available to analyses on a best-effort basis; this is obviously possible for Jimple generated from Java source code, but more difficult when generating Jimple from Java bytecode, due to the required local variable manipulations in creating Jimple from stack code.

[1] Many uses of Soot are documented at: https://svn.sable.mcgill.ca/wiki/index.cgi/SootUsers.

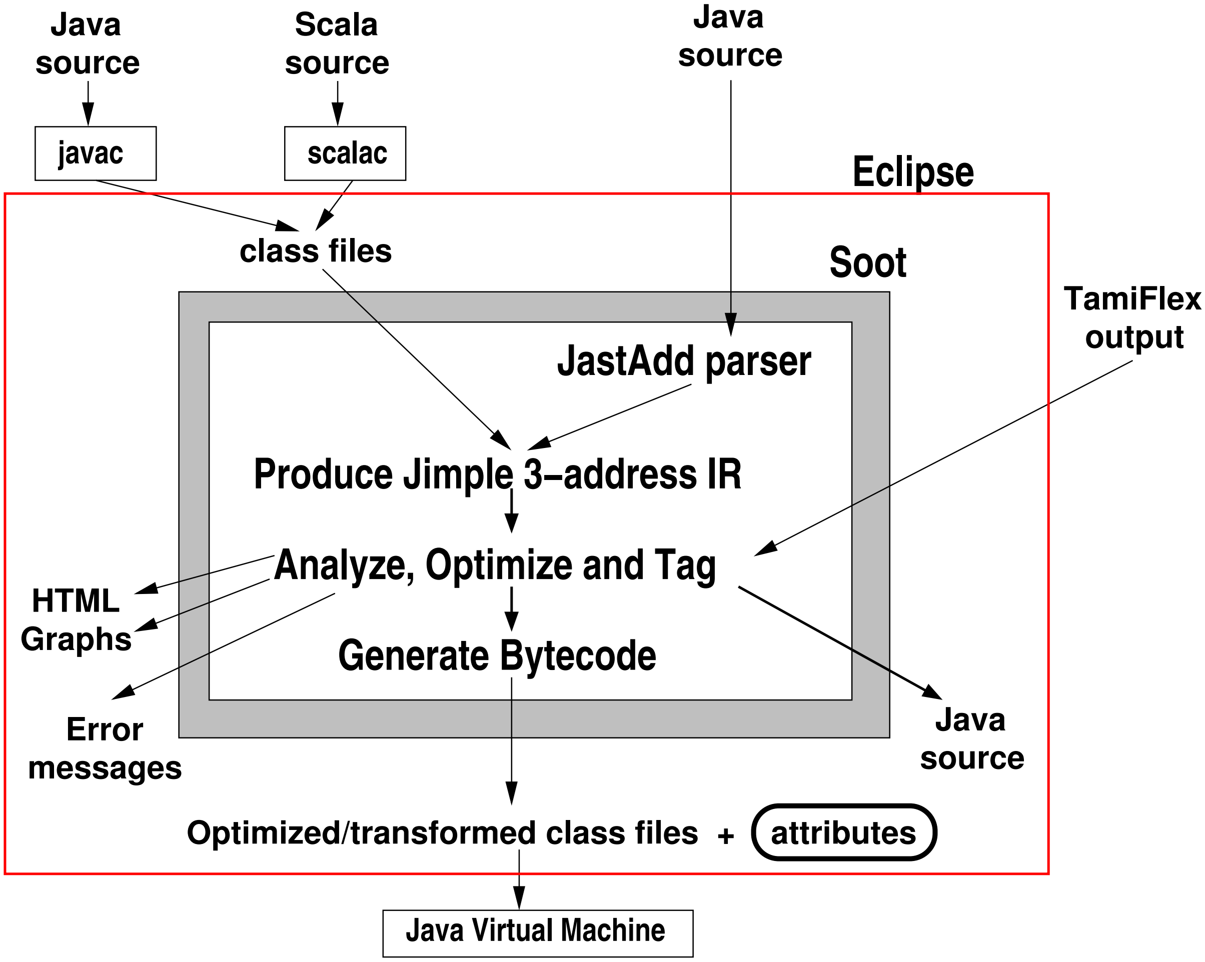

Fig. 1. Overview of Soot workflow.

A key feature of the Soot framework is its support for implementing intraprocedural data-flow analyses, a central technique in compilers research. Soot users can create a data-flow analysis by specifying the abstraction and implementing transfer functions for that analysis. The Soot framework also provides useful data structure implementations for common abstractions.

Many Soot users have implemented data-flow analyses. Some examples include an analysis to statically approximate heap reference counts and thus generate annotations informing the runtime system about objects that are safe to deallocate [CR06]; an analysis on constraint graphs to find unnecessary array bounds checks [QHV02]; and an analysis to evaluate the adequacy of test suites for database-driven applications [KS03].

B. Interprocedural Features

Sophisticated static analyses require call-graph and pointer information, which is fundamentally an interprocedural problem. Soot includes the Spark pointer analysis toolkit [LH03], [Lho02], and also supports the BDD-based PADDLE pointer analysis framework, which adds context-sensitivity.

Call Graphs. The Soot framework provides call graph information. Static analyses (particularly interprocedural ones) need to know, for each call site, all possible targets of that call site. Java's object-oriented nature complicates this problem, since the identity of the callee, or target, depends on the run-time type of the receiver object for virtual calls, and most calls in Java programs are virtual calls.

Spark's call graph construction algorithms compute an over-approximation of the set of calls that may occur at runtime. Call graph edges connect sources, which are represented as (method, statement) pairs, and targets, which are always the first statement in a method. Overapproximation means that any call which could occur in any execution of the program must appear in the call graph.

Spark implements a number of different call graph construction methods, including Class Hierarchy Analysis and Rapid Type Analysis; however, we found that the most effective call graph construction method proceeds on-the-fly and builds the call graph at the same time as it computes points-to sets, starting with the CHA call graph as an initial approximation for the set of reachable methods.

The resulting call graphs can be queried by callsite; by calling method; or "backwards", by target method. For efficiency reasons, Spark returns iterators as the results of callgraph queries. Spark also makes the set of transitive targets for any method or statement available to clients. Finally, Spark also exposes the set of reachable methods for a program—the set of methods transitively reachable from the program's entry points (which include class initializers, etc., in addition to the program's `main()` method).

Pointer Analyses. Many static analyses require pointer information: they need to know if two variables p and q may refer to the same heap object at runtime. Spark implements a context-insensitive subset-based points-to analysis (Andersen-style, in the notation of [HP00]). An analysis may query a points-to analysis and obtain an abstraction of the objects pointed to by a local variable or object field, represented by a `PointsToSet`. The `PointsToSet` supports two queries: 1) the set of possible types for the objects in that set; and 2) whether two `PointsToSet` objects have non-empty intersection. The possible types give information about possible receivers for method calls to the variable whose values the points-to set models. If variables p and q have points-to sets with non-empty intersection, then they may point to the same object.

Side Effects. Soot includes a side effect analysis, which builds on top of (any Soot-based) points-to and call graph analyses. The side effect analysis can determine whether a statement s has a possible dependence on statement s'. It works by defining read-write sets for each statement and then comparing their intersection; two statements with disjoint read-write sets are independent, while a statement that writes a value followed by another statement that reads the same value follow a dependency relation.

Alternatives and Extensions to Spark. While Spark is the basic call graph and pointer analysis producer for Soot, it is not the only option. Manu Sridharan has contributed a demand-driven pointer analysis [SGSB05], which can answer non-empty-intersection queries for variables using limited amounts of context-sensitivity; this analysis is part of the main Soot distribution. PADDLE [LH08a], [Lho06] provides a BDD-based context-sensitive pointer analysis; however, it requires extra dependencies, both at build time (because it is written using a domain-specific language [LH08b]) and at run time (to efficiently compute on the binary decision diagrams).

While the pointer analyses we have described so far are all may-alias analyses, Soot also contains an intraprocedural must-alias analysis. Object representatives [BLH08b] integrate this must-alias analysis with both intraprocedural and whole-program may-alias analysis; Spark is one source of whole-program may-alias analysis.

Reflection. Many Java programs use reflection to access classes or their members by name. Such reflective accesses are hard to analyze statically: class and method names can be computed at runtime or loaded from files that the static analysis does not have access to. One partial workaround is to record information about reflection usage at runtime (during training runs) and to use this information when performing call-graph and points-to analysis. The latest version of Soot integrates with the TamiFlex tool chain [BSS+11], which follows the described approach.

C. Soot Output

Soot supports a number of mechanisms for making analysis results available. Developers' options include: 1) outputting transformed class files; 2) outputting error messages; 3) generating HTML or graphs containing analysis results; or 4) creating (potentially transformed) class files annotated with results obtained from program analysis.

An example of a tool that uses Soot to transform classfiles is the DUSC tool by Orso et al [ORH02]. This tool implements dynamic software updating for Java programs by first carrying out a semantics-preserving transformation of an application to enable dynamic updating, and then creating Java classes for new versions of the application, which can be swapped in at runtime.

Soot has been used as the backend of the AspectBench `abc` compiler for AspectJ [ACH+06]. In this context, Soot performs the necessary bytecode transformations (weaving) as specified in a program's aspect declarations. The Jimple three-address code representation helps simplify the weaving process. Furthermore, Soot can clean up the naïve code generated as output from weaving, both using its standard optimizations as well as through specialized optimizations for aspect-oriented programs [ACH+05], mitigating the burden on Java Virtual Machines as they run AspectJ programs.

Soot can also implement non-semantics-preserving transformations. Tkachuk and Dwyer [TD03] used Soot to generate non-executable summaries of a system environment's behaviour, simplifying subsequent model checking of the system. The generated environment summaries include all of the possible effects of the actual environments, and are expressed using Java bytecode augmented with special modeling primitives (written using method calls).

Error messages are appropriate when a Soot-based tool is looking for violations of safety properties; for instance, the tool described in [BLH08a] statically detects likely tracematch violations and reports them to the user. This tool also works with the `highlight` tool[2] to generate annotated Java source code in HTML format, enabling users to browse these violations and to visualize the relevant finite-state machines.

Attributes are, in practice, most useful for viewing statement-level flow analysis results (using the Soot Eclipse plugin [LLH04]). We initially incorporated attributes to communicate static analysis information to runtime systems [PQVR+01]. Attributes are especially useful for communicating the results of expensive analyses, like side-effect information [LLH05], and we can indeed encode this information in attributes.

[2] http://www.andre-simon.de

III. Working with and Extending Soot

It is possible to interact with Soot at many levels. We designed Soot such that the typical user would add some passes to Soot and then run their augmented version of Soot.

Running Soot. Simply running Soot on the command line is the simplest way to interact with Soot. Jimple code is useful for humans as a cleaned-up version of Java bytecode. In fact, we created the Jimp dialect of Jimple specifically for this purpose; it sacrifices completeness for readability, e.g. by omitting full names of fields. Full Jimple can be recompiled into Java bytecode, while Jimp is most useful for manual inspection.

Soot can also optimize bytecode. However, modern Java virtual machines and just-in-time compilers do an excellent job of optimizing code (using information available only at run time), so that Soot's optimizations do not improve the performance of normal bytecode. The optimizations are useful for non-standard bytecode, such as that generated by the abc AspectJ compiler.

Another way of running Soot is through its Eclipse plugin [LLH04]. The Eclipse plugin allows users to invoke Soot (or augmented versions thereof). More importantly, however, it also allows users to view Jimple CFGs and static analysis results. In particular, the Soot Eclipse plugin enables developers to see flow analysis results as they are being computed. Also, the plugin can display attributes summarizing analysis results.

Building on top of Soot. To use Soot for program analysis research, users must write compiler passes extending Soot's functionality. Soot provides two basic types of passes: `BodyTransformers` and `SceneTransformers`. The `BodyTransformer` is most suitable for intraprocedural analyses, and is executed on each method in a program. The `SceneTransformer` executes once and may analyze and manipulate the entire program as it sees fit. These `Transformers` belong to `Packs`, which correspond to collections of compiler passes. We have documented the set of `Packs` that Soot ships with out-of-the-box, and researchers are free to add to a provided `Pack`, if it runs at the appropriate phase, or to create their own `Packs`.

Different `Transformers` may communicate by sharing `Maps` containing analysis results, or by using attributes. We explicitly chose to disallow analysis passes from storing information on IR statements (for instance, by subclassing Jimple `Stmts`), as that makes it difficult to compose different analyses.

We recommend that users add passes to Soot by creating their own custom main class which manipulates Soot's `Packs` by adding appropriate `Transformers`, and then invoking the Soot `main()` method. Soot's Eclipse plugin can be used to generate templates for this use case (File→New→Example). It is also possible to manually call the different methods that Soot's `main()` method invokes in one's own code. While this is a more flexible way of using Soot, we do not recommend it for most users.

While the primary way to extend Soot is by adding new analyses, it is also possible to add new intermediate representations and output formats; the SSA-based intermediate representation Shimple [Uma06] is one such example.

Unfortunately, we failed to include a good mechanism for research groups to share their Soot extensions (without getting them checked into the main Soot distribution). We return to this topic in Section VI.

IV. Soot Development Process and Community

Soot was originally developed by Raja Vallée-Rai for his MSc thesis [VR00]. The initial development took place during the initial surge of interest in analyzing Java. This happened in 1999–2000 and culminated in the release of Soot 1.0, a viable intraprocedural Java analysis framework. Soot confirmed that it was possible to carry out sophisticated analyses of Java programs, starting from the bytecode (rather than source code) for these programs. Since its initial release, Soot has added numerous features, as previously described in Section II.

Often, important features were added to Soot and then refined by later contributors. Two examples are: 1) the local variable type inference system, which first used a technique by Gagnon et al [GHM00], and was later replaced by Bellamy et al's fast type inference [BAdMS08]; and 2) call graph information, which was initially prototyped by Sundaresan et al using Variable Type Analysis [SHR$^+$00], and later subsumed by Spark (and PADDLE, to some extent).

Soot's core development has centered around the Sable lab at McGill University. The role of primary Soot maintainer has rotated around a number of McGill students and alumni. Soot maintenance involves integrating external patches, responding to questions on the Soot mailing list, and periodically releasing new versions of Soot. Soot's continuing development would surely have benefitted from a full-time systems programmer, but we did not have the resources for such a position.

We have integrated significant external contributions into Soot. Notable external contributions include a fast type assigner for local variables by Ben Bellamy [BAdMS08], a JastAdd-based parser for Java 5 source code by Torbjörn Ekman [EH07], and a demand-driven pointer analysis by Manu Sridharan [SGSB05].

A. Support and Community

The most active meeting place for the Soot community is its mailing list, which receives about 30 messages in an average month, with some spikes up to 120 messages. The list is the primary support channel for Soot users. Most questions are answered promptly by Soot developers, and the publicly-available mailing list archive is a useful resource for Soot users. In a laudable trend, we have noticed that the Soot community has evolved to the point that Soot users sometimes respond to each others' questions on the Soot mailing list.

We have experimented with other collaboration tools, but they have generally been less successful than the mailing list. A public bug-tracking system (Bugzilla) exists, as well as a wiki. The wiki is useful for recording certain specific types of information, such as a list of Soot users, but does not build a sense of community. The bug tracking system can record Soot bugs; however, the Soot team does not generally have the resources to fix non-critical Soot bugs. We always welcome bug reports that come with patches. Such bugs are most likely to be fixed quickly.

We have licensed Soot under the GNU Lesser General Public License (LGPL). This license is appropriate for many, but not all, users; it is permissive enough to allow Soot users to incorporate Soot into arbitrarily-licensed client code, but can trigger alarm from overly-conservative institutional lawyers. We understand that some additional users might have been able to use Soot had it been released under a more permissive license, such as the Apache or BSD licenses. As a strong supporter of free software (in the GNU sense), Raja selected the GNU LGPL as the Soot license; future authors of compiler frameworks ought to consider which license is most compatible with their ideology and their desire for the adoption of their software. Note that McGill's Sable group will release the McLab compiler framework for MATLAB [CLD$^+$10] under the Apache 2.0 license.

Soot's Subversion repository is publicly readable. While the core Soot code has not attracted external committers, we have welcomed documentation commits from the authors of the Soot survivor's guide, described below. A server produces nightly builds based on the contents of the repository.

During its early development, Soot came with a test suite verifying its functionality and testing for regressions. This test suite has been lost to the mists of time. Our current verification strategy for Soot is to ensure that it passes the `abc` tests, which exercise significant portions of Soot's functionality.

B. Documentation

Soot has extensive documentation in a variety of formats. Documentation is critical to Soot's usefulness as a research tool, and we (and others) have expended significant effort to create documentation for Soot. (Documentation is an obvious potential use case for the Soot wiki.)

The most basic form of documentation is the API design. The first author recalls extended discussions with Raja about Soot API design. We believe that the core internal Soot API is reasonably self-documenting and easy to use. Soot also has some Javadoc documentation comments to elucidate the API, but it is quite tedious to provide complete Javadoc coverage. Unless such documentation was somehow crowdsourced, it would not be reasonable to expect research compiler frameworks to come with complete Javadoc documentation comments. We believe that we have provided the best that one could hope for in a research compiler: good API design coupled with occasional Javadoc comments.

The Soot team has also created a set of documents explaining how to carry out various tasks using Soot. These documents walk the reader through tasks such as implementing dataflow analyses and adding attributes to Java class files. Also, the Soot community has contributed additional documentation, notably in the form of the excellent Soot Survivor's Guide by Einarsson and Nielsen [EN08].

One form of Soot help that we would like to point out lies in error messages. Two common tripping points are the `OutOfMemoryError` and incomplete-classpath errors. Out-of-memory errors occur because Java virtual machines' default memory allocation is insufficient for running whole-program pointer analyses. Soot therefore catches this error and displays an error message telling the user how to increase the memory allocation. The incomplete-classpath error occurs because Soot needs all of a program's libraries to carry out pointer analysis; when it cannot find needed classes, it recommends that the user include the `jce.jar` and `jsse.jar` files, which are most likely to be missing.

A final form of Soot documentation has been half-day tutorials at the PLDI and CASCON conferences. We have summarized the contents of these tutorials in the present paper, but the tutorial slides provide full details about how to actually carry out required tasks using Soot. In particular, the Soot tutorials explained Soot's basic structure; how to implement intraprocedural analyses and to add these analyses to Soot; how to use the call graph and pointer analysis information produced by SPARK; and how to add attributes to code outputted by Soot. While the tutorials were not heavily-attended, we believe that they helped some aspiring Soot users, and the tutorial slides probably helped more Soot users.

V. Soot's Past Evolution; Future Work on Soot

This section explains relevant major changes and extensions that we have made to Soot in the past, and proposes future extensions to Soot. We would gladly welcome any external contributors that are interested in working on these extensions. Soot has also incorporated hundreds of minor improvements, which are not mentioned here.

Singletons and multiple Soot runs. Soot's initial design used the Singleton design pattern in a number of places. This was quite inconvenient for users who wished to invoke Soot multiple times from their own client code. We refactored Soot to eliminate most of the singletons and global variables, consolidated the remaining global variables in a `G` singleton, and implemented a static analysis to detect singletons and globals that did not live in `G`. We also added the `G.reset()` method to reset Soot's state.

Partial programs. Soot requires the entire program to compute sound results for whole-program analyses, such as pointer analysis. However, many applications (particularly Eclipse-based software engineering ones) do not have or need the whole program. Using an external tool by Dagenais [DH08], it is possible to parse Java source code for incomplete programs. We also recently fixed Soot's support for analyzing Java bytecode without all referenced libraries; Soot will warn the user that the results are probably unsound, but it will then continue the analysis on a best-effort basis.

Java front-end parsers. Soot originally did not include a Java front-end parser. Fortunately, this problem attracted quite a bit of interest, and Soot gained both a Polyglot-based [NCM03]

Java front-end, and later on, a Java 1.5-compatible JastAdd-based front-end, as previously mentioned.

Increased efficiency. On a demand-driven basis, we have improved the performance of selected parts of Soot. We have noticed that it would have been impossible to predict which parts should have been optimized from the outset. Soot has benefitted from improved implementations of the class hierarchy, strongly-connected components, local type inference, and local defs/uses calculators.

A. Future Directions for Soot

We identify three directions for future improvements on Soot: faster startup and computation time; interprocedural analysis support; and support for Java language extensions.

Any analysis which uses whole-program analysis results must wait for Soot to parse thousands of classfiles. The problem is that even the smallest Java program contains dependencies on the Java class library, which has extensive cross-references between library classes. The result is that, once an analysis calls for pointer analysis results, Soot's running time shoots up from under 10 seconds to over a minute. We believe that it would be worthwhile to serialize the generated Jimple code for the Java class libraries once and for all, perhaps using the techniques of Gligoric et al [GMK11]. This should enable much faster startup time for Soot analyses. Performance improvements could also come from rewriting the Jimple creation code and from using multiple threads to create Jimple code, which has been a nice-to-have project since 2000.

While Soot has excellent support for intraprocedural analyses, developers of interprocedural analyses are left much more to their own devices. As described above, Soot provides the `SceneTransformer` for authors of novel interprocedural analyses. Unfortunately for such authors, the `SceneTransformer` does not provide any help with structuring an analysis. In particular, a Soot user needs to figure out how to traverse the analyzed program's classes and how to combine the analysis results from different methods. Some design work in defining the common case and making it easy to program could help a lot of analysis authors.

Finally, it has historically been difficult to evaluate language extensions (such as type system extensions) using Soot. While Soot carries out extensive type inference on bytecode, it has not been easy to get source code annotations (e.g. type annotations) into Soot's IRs. Such research should now be easier using the JastAdd front-end; however, we are not aware of any projects investigating Java language extensions where Soot analyzes programs given in an extended Java source language.

VI. Reflections on Soot

In this section, we mention some difficulties we had in developing Soot, suggest desirable features for future compiler frameworks, and conclude with some reflections on reasons for Soot's success.

Our experience is that, in the large, Soot now does what we thought it would do. One unexpected application of Soot was for unsound and incomplete program analyses. When we were initially designing Soot, such analyses were unknown in the research community; however, in the intervening 12 years, they have become quite popular. Soot can implement such analyses without any problems.

Difficulties. We would like to highlight two difficulties: keeping Soot up to date in the presence of external changes, and encouraging Soot users to contribute their changes.

The Java language has changed heavily since the initial release of Soot; Java 1.3 was the newest version of Java when Soot 1.0 was released in 2000, and it did not include generics, `invokedynamic`, annotations, or `foreach` loops, among other changes. Fortunately, the changes to the virtual machine have been more limited. Nevertheless, it has been difficult for the Soot team to keep up with the changes in the Java virtual machine (for bytecode inputs) and particularly in the Java language (for the Java front-end). Changes to Eclipse have also been a (much more serious) problem for the Soot Eclipse plugin.

While we have highlighted a number of contributed changes in this paper, we would have liked to incorporate many more Soot contributions from non-McGill users. This is particularly true given that others have developed Soot extensions that would be of general interest.

Ways to Improve. We have discussed some Soot-specific potential improvements in Section V. In this section, we discuss ideas for improving compiler frameworks that we believe would be of interest to the broader community.

We believe that there are several reasons for Soot extensions not getting merged back into the main Soot code. First, compiler frameworks should be designed to make it easier to independently release framework extensions. While the Eclipse plugin system may be needlessly complex for a research compiler, it may still be useful to have a simple extensions system. Soot could be much-improved in this area, as we did not think of this problem at the time. Second, releasing software is time-consuming and unrewarding. Conferences should value software and data releases more highly when evaluating papers; they are an integral part of the scientific process. Some conferences, such as ESEC/FSE, are starting to encourage the release of more complete information along with papers, and we applaud this trend.

We have also noticed that it is difficult to publish framework papers; there is no real refereed paper which describes Soot as a system. In Soot's case, the best paper to cite is [VRGH$^+$00], which describes how Soot converts Jimple back into bytecode. We encourage conferences to accept more framework papers.

On a more technical level, Soot recomputes a lot of data between passes. It has to recompute the data because it does not know how much of the data was invalidated by intermediate computations. Incremental [BLS$^+$11] or reactive [DFR11] computation would help improve the performance of compiler frameworks.

Finally, quasiquoting is an extremely useful feature for compiler frameworks. In this context, quasiquoting [Baw99] allows writers of transformation passes to specify code templates (and the relevant values to populate those templates) for inclusion in the outputted code. It is certainly quite inconvenient to manually create Jimple AST nodes in a transformation pass. We were not aware of quasiquoting while we were initially designing Soot, but it could easily be added at any time.

Reasons for Success. We believe that Soot succeeded because it provided the right features at the right time and was easy enough to use. We discussed Soot's features in Section II. The most important features include: 1) Soot's support for Java; 2) the handy Jimple intermediate representation; and 3) the Spark pointer analysis toolkit. The pointer analysis was especially important, because most nontrivial analyses of Java code must soundly reason about the behaviour of pointers. Any compiler framework is going to be somewhat difficult to use, but it seems that Soot was usable enough, given sufficient determination. Ease of use also includes Soot's software license, nightly builds and preparation of occasional Soot releases (incorporating patches from the community), and responsiveness on the Soot mailing list.

Acknowledgment. Soot's development was supported in part by Canada's Natural Science and Engineering Research Council, the Fonds de recherche du Québec—Nature et technologies, IBM's Centre for Advanced Studies, and an Eclipse Innovation Grant. Eric Bodden is supported by CASED (www.cased.de).

We would like to thank all of the contributors who helped in the development of Soot, and we would like to especially note Raja Vallée-Rai's pioneering work on Soot.

References

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128, 2005.

[ACH+06] Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: an extensible aspectJ compiler. *Transactions on Aspect-Oriented Software Development I*, 3880:293–334, 2006.

[BAdMS08] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. In *OOPSLA'08*, pages 475–492, 2008.

[Baw99] Alan Bawden. Quasiquotation in Lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, January 1999.

[BLH08a] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 36–47, November 2008.

[BLH08b] Eric Bodden, Patrick Lam, and Laurie Hendren. Object representatives: a uniform abstraction for pointer information. In *Proceedings of the 1st International Academic Research Conference of the British Computer Society (Visions of Computer Science)*, pages 391–405, 2008.

[BLS+11] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, October 2011.

[BSS+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11: International Conference on Software Engineering*, pages 241–250. ACM, May 2011.

[CLD+10] Andrew Casey, Jun Li, Jesse Doherty, Maxime Chevalier-Boisvert, Toheed Aslam, Anton Dubrau, Nurudeen Lameed, Amina Aslam, Rahul Garg, Soroush Radpour, Olivier Savary Belanger, Laurie Hendren, and Clark Verbrugge. McLab: an extensible compiler toolkit for MATLAB and related languages. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 114–117, New York, NY, USA, 2010. ACM.

[CR06] Sigmund Cherem and Radu Rugina. Compile-time deallocation of individual objects. In *Proceedings of the 2006 International Symposium on Memory Management (ISMM 2006)*, Ottawa, Ontario, Canada, June 2006.

[DFR11] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, October 2011.

[DH08] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 313–328, New York, NY, USA, 2008. ACM.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[EN08] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to Java program analysis with soot. http://www.brics.dk/SootGuide/, July 2008. version 1.1.

[GHM00] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[GMK11] Milos Gligoric, Darko Marinov, and Sam Kamin. CoDeSe: Fast deserialization via code generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 298–308, July 2011.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 113–123, New York, NY, USA, 2000. ACM.

[KS03] Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 98–107, New York, NY, USA, 2003. ACM.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[LH08a] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008.

[LH08b] Ondřej Lhoták and Laurie Hendren. Relations as an abstraction for BDD-based program analysis. *ACM Trans. Program. Lang. Syst.*, 30(4):1–63, 2008.

[Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[Lho06] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[LLH04] Jennifer Lhoták, Ondřej Lhoták, and Laurie Hendren. Integrating the Soot compiler infrastructure into an IDE. In E. Duesterwald, editor, *Compiler Construction, 13th International Conference*, volume 2985 of *LNCS*, pages 281–297, Barcelona, Spain, April 2004. Springer.

[LLH05] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using interprocedural side-effect information in jit optimizations. In R. Bodik, editor, *Compiler Construction, 14th International Conference*, volume 3443 of *LNCS*, pages 287–304, Edinburgh, April 2005. Springer.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[ORH02] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of Java software. In *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM'02)*, pages 649–658, Montreal, Canada, October 2002.

[PQVR+01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, pages 334–554, 2001.

[QHV02] Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 325–342. Springer, 2002.

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[SHR+00] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 264–280, 2000.

[TD03] Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of ESEC/FSE-11*, pages 188–197, September 2003.

[Uma06] Navindra Umanee. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, February 2006.

[VR00] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, July 2000.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.