Persistent Analysis Results [1]

Adrian Prantl, Jens Knoop and Markus Schordan

Institut für Computersprachen
Technische Universität Wien
email: {adrian, knoop}@complang.tuwien.ac.at
and
University of Applied Sciences Technikum Wien
email: schordan@technikum-wien.at

Abstract In this article we propose a new approach for storing the results of static program analyses to make them persistent and accessible for future use by both application programmers and tools. There are two key ideas. First, to attach the analysis results to the source code instead of the intermediate representation or the control-flow graph; we call this *behaviour-carrying code*. Second, to prepare this code to make analysis results automatically checkable by both static and dynamic verifiers. This opens up multiple new applications for program and tool development for compiler developers and more generally software engineers. The approach has been implemented in the *SATIrE* source-to-source analysis framework using the *Termite* program transformation library. First experiments show encouraging results and proved already valuable to increase the quality of analyzer implementations.

1 Motivation

Static program analysis aims at computing information about the runtime behaviour of a program without executing it. Theoretically well-founded are approaches based on data-flow analysis and abstract interpretation [Kil73, CC77, Hec77, Muc97]. Most commonly the information computed for a program point holds either *universally*, i.e., for every program execution reaching this point, or *existentially*, i.e., for some program execution reaching it. On the algorithmic side, this corresponds to *must* and *may* program analyses, respectively. *Available expressions* and *reaching definitions* are typical examples of properties computed by *must* and *may* analyses, respectively:

[1] This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, *Compiler Support for Timing Analysis* ((*CoSTA*), `http://costa.tuwien.ac.at/`, and the Commission of the European Union within the 7th EU R&D Framework Programme under contract No 215068, *Integrating European Timing Analysis Technology* (*ALL-TIMES*), `http://www.mrtc.mdh.se/projects/all-times/`.

– Available Expressions: An expression is computed along *every* path reaching a particular program point (must-information)
– Reaching Definitions: A definition of a variable is valid along some program path reaching a particular program point (may-information)

Results of analyses like for available expressions or reaching definitions are the basis of optimizing program transformations applied by a compiler. For example, variables computed to have a unique constant value at runtime can be used to replace expressions by their values; a transformation known as *constant folding*. Optimizing compilation, however, is not the only domain where results of program analyses are valuable. Another important domain concerns non-functional program properties such as resource consumption in terms of time or memory usage. Programmers often need to get a better grip on such run-time characteristics of the completed software, and thus need to get information on a program's spatial and temporal worst-case behaviour.

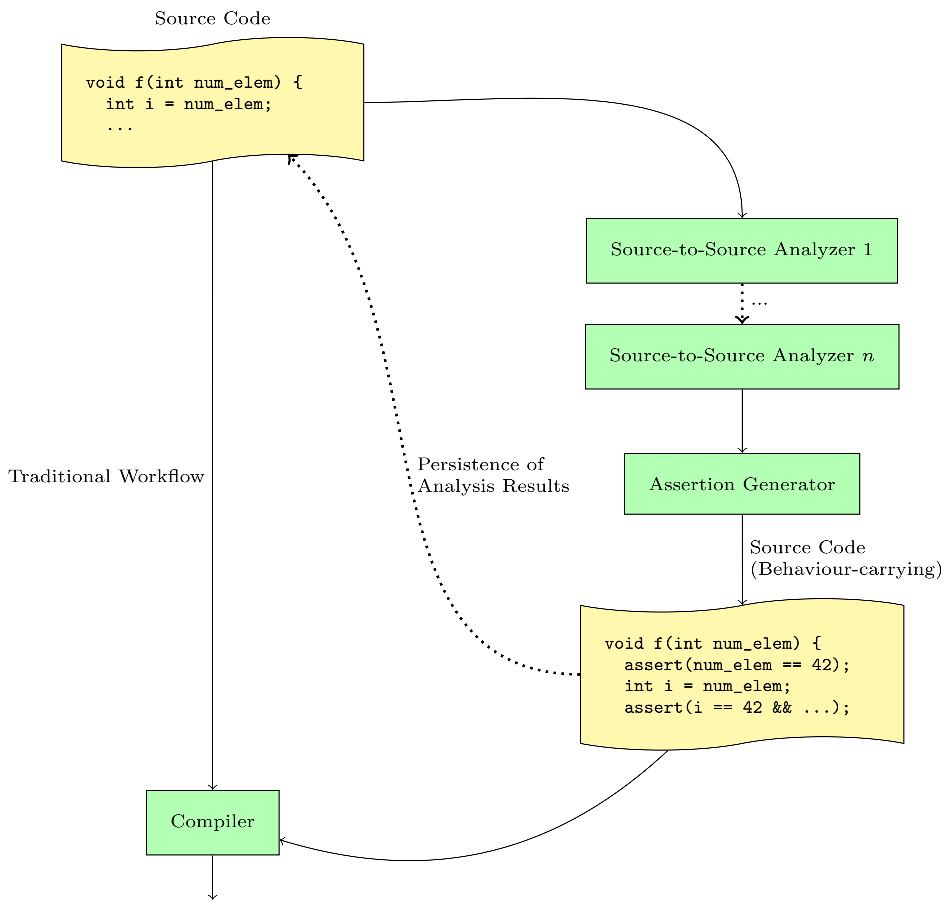
These days, the fields of program optimization and resource analysis are essentially served by different communities – often with little sharing between. For example, in the field of resource analysis a broad range of analyzers exists, but few are integrated with a compiler. One reason for this separation might be the data formats used to store analysis results. In a typical compiler, analysis results are temporary information which are calculated on demand for the purpose of a particular optimization. For this reason, they are usually attached to the control-flow graph (CFG) of a program and are thus not easily visible to the outside world. While adequate and efficient for the purpose of optimizing compilation, this has the drawback of impacting the portability of these results in order to make them amenable for other uses. This of course hurts the goal of interoperability between various tools. Next we present our approach to improve on this situation.

2 The Source-to-Source Workflow

In principle, a source-to-source program analyzer can be viewed as a portable stand-alone component that can be plugged before an off-the-shelf compiler system and that outputs analysis results in some external format (cf. [Sch08]). The resulting workflow is illustrated in Figure 1:

1. Perform a (potentially complex and expensive) analysis to some source program.
2. Attach the analysis results to the source code itself.

This way the analysis results become explicitly visible for the user. We call this *behaviour-carrying code* (*BCC*), in analogy to *proof-carrying code* (*PCC*) introduced by Necula and Lee [Nec97, NL98], which has a similar flavour. Besides informing the user about certain aspects of the run-time behaviour of the program, the BCC can be subject to other analyzers extending the behaviour information further. Moreover, at any step, the BCC can be passed on to the


Source Code
Traditional Workflow
Source-to-Source Analyzer 1
...
Source-to-Source Analyzer n
Persistence of Analysis Results
Assertion Generator
Source Code (Behaviour-carrying)
Compiler

```
void f(int num_elem) {
  int i = num_elem;
  ...
```

```
void f(int num_elem) {
  assert(num_elem == 42);
  int i = num_elem;
  assert(i == 42 && ...);
```

Figure 1. Source-to-source analysis work flow

compiler to proceed with compiling it and to generate executable code. Transformations applied by the various stages of the compiler, in particular its optimizer stage, can directly refer to this information without need to (re-) compute it; thus speeding-up the compilation process.

The feedback of the BCC of a source program into the workflow of the source-to-source analyzer is illustrated in Figure 1, too, making up the third and fourth step of the complete workflow:

3. Feed the BCC of a source program to the source-to-source analyzer obtaining a new BCC with extended behaviour information.
4. Stop analyzing the BCC and pass it on to the compiler.

Considering constant propagation as an example Figure 1 suggests another application mode, which generally leads to stronger analysis results. This requires that the source-to-source analyzer can operate across the boundaries of program modules. This way, it can benefit by checking and using the information on variables with constant values of used (and already analyzed) modules for strengthening the analysis results for the module currently under consideration.

"Plain" BCC is already useful to inform the user on certain aspects of the run-time of a program and can easily be reused and exploited by other program analyses typically applied throughout the compilation process. Next we demonstrate how to prepare plain BCC in order to make analysis results automatically checkable, both by static and dynamic verifiers. This opens up multiple new applications as we illustrate by three examples subsequently.

3 Checkable Assertions

Intuitively, the results of a program analysis provide information about the possible states of the underlying program at the various program points. Considering the two points immediately before and after the execution of a statement, this information can be thought of in terms of a pre- and a postcondition describing a property of the program states which may be valid immediately before and after executing this statement. Using the analogy of a (valid) *Hoare assertion* [Hoa69],

$$\{P\}\ Q\ \{R\}$$

where P is the precondition of a program Q and R a postcondition capturing the effect of Q wrt P, P and R are *assertions* which ensure a certain property of program Q. In the same fashion also the results of a program analysis attached to a statement can be considered assertions about the program behaviour at this statement. Some programming languages offer an integrated concept of assertions such as C in terms of *assert()* macros or Eiffel in terms of *contracts*. These language-provided assertions can be automatically checked at run-time and are useful for security checks. In the following we present our approach of how to present analysis results such that they receive the flavour of assertions, which can then be subject of static and dynamic verifiers.

To this end we define a program transformation that modifies the analysis information attached to a program such that the attached analysis information can be used like an assertion. It is worth noting that this does not affect the semantics of the program while offering the advantage of having the analysis information immediately available.

Conceptually, this program transformation can be thought of to work by transforming the analysis information attached to a program point into an assertion. An implementation might be organized to directly output assertions and skipping the intermediate step of outputting plain analysis information. In our approach, we distinguish assertions addressing *universally* and *existentially* valid program properties, respectively. Using the C language as source language for demonstration, we will show next how to express the two kinds of assertions for this setting.

We use the following notational conventions: P denotes the result-predicate from the ideal analysis, $\mathcal{M}_P$ the set returned by the implemented analyzer and L the location of interest in the analyzed program.

3.1 Assertions for Universally Valid Information

Intuitively, an information is *universally valid* at a location L if it is valid on every path reaching L. Program analyses computing universally valid information are commonly called *must*-analyses. Representing universally valid information of an analysis in terms of an assertion requires that the assertion specifies a test for the respective analysis information. This is illustrated by the following example:

Example 1. Constant propagation (a *must*-analysis) yields universally valid information in terms of expression-constant pairs: $P(exp, const) : exp = const$. In the C programming language, an assertion expressing this information can take the form `assert(exp == const)`.

Following [ALSU07], an analysis result is *safe* if it does not report any false positives. Using the notion of safety, the above example can be generalized for any universal information as summarized in Observation 1. Intuitively, this observation follows immediately from the definition of safety and the fact that an assertion `assert(`$\mathcal{M}_P$`)` will fail, if the result is not safe.

Observation 1. If an analysis reports a set $\mathcal{M}_P$ to universally fulfill a predicate P at location L, a statement `assert(`$\mathcal{M}_P$`)` inserted at L will evaluate to *true*, iff $\mathcal{M}_P$ is *safe*.

3.2 Assertions for Existentially Valid Information

Intuitively, information is *existentially valid* at a location L if it is valid on some path reaching L. Program analyses computing existentially valid information are commonly called *may*-analyses. Since existentially valid information need not hold along all paths, an assertion addressing such a property cannot restrict itself to simply specify a test for it (obviously, the result *can* be false). Instead, it must specify a test for the precision of the analysis information. The following example illustrates this idea.

Example 2. A *may*-alias analysis yields sets of potentially aliasing pointer variables: $P(p_1, ..., p_n)\colon addr(p_i) = addr(p_{i+1}),\ i \in \{1, \ldots, n-1\}$. An assertion has the form `assert(q1 != q2 && q1 != q3 && ... &&` q_{n-1} `!=` q_n`)`, where $q_i \neq p_j$ are all remaining pointer variables. For a program with k pointer variables and an analysis result containing j aliasing variables, the assertion consists of $n * (n-1)/2, n = k - j$ comparisons.

We call an analysis result *precise* if there are no other solutions. This means that any combination of parameters which is not part of the analysis result can not be in P. Putting all this together, leads us to our second observation, which, intuitively, follows from the definition of precision and the reasoning that the analysis result includes all solutions which can be seen as follows: For all tuples $\bar{x}$ that are not in the analysis result P is not satisfied. $\forall \bar{x} \notin \mathcal{M}_P\ .\ \neg P(\bar{x}) \iff \forall \bar{x} \in P . \bar{x} \in \mathcal{M}_P \iff P \subseteq \mathcal{M}_P$.

Observation 2. If an analysis reports a set $\mathcal{M}_P$ to existentially fulfill a predicate P at location L, a statement `assert(`$\forall \bar{x} \notin \mathcal{M}_P\ .\ \neg P(\bar{x})$`)` will evaluate to *true*, iff $\mathcal{M}_P$ is *precise.*

We have now shown how to transform the program to explicitly contain meta-information about itself that was brought forward by an analysis.

4 Applications for behaviour-carrying code

In Section 2 we have already seen how the source-to-source workflow allows for a more flexible interaction of analysis tools and compilers. By interlacing the analyzed information with the source code the subsequent system compiler can take advantage of new analyses without the necessity to re-implement them. Behaviour-carrying code also serves to make the analysis results persistent. By encoding analysis results in the source code they can be stored for future use in a universally recognizable data format. Re-using this information then only requires to read this information and to run the analysis again in order to re-transform the external format into the internal one. It is worth noting that this does not require a fixed point iteration rather a single visit of each node. In many cases even a weaker analysis will suffice, since the information is now available in a more explicit fashion.

In this section we present two applications of behaviour-carrying code – one targeted at software engineers and one targeted at compiler developers.

4.1 Behaviour-carrying code for the end-user

Bring forward informal interfaces. Behaviour-carrying code in its “plain” incarnation is not only useful in the program analysis and compilation work flow but can also prove a valuable tool for the software engineer. With the help of BCC, the software engineer can get a clearer picture of large software projects. It can serve as *documentation* and *code understanding* tool. Often engineers are confronted with the maintenance of complex legacy systems that are not fully understood and/or documented. When faced with the task to add a new feature to such a system it is crucial to know the behaviour of the existing system beforehand. We believe that behaviour-carrying code can be a valuable means for documenting a *specification* extracted from legacy projects via static analyses.

Freeze interfaces against unwanted modifications. This idea can be further refined in the respect of fully behaviour-carrying code. We can use a static program analysis to reveal invisible interface conventions and to use the BCC as a means to future-proof the interface. Figure 2 shows an example how this could be applied in practice.

```
enum e_types classify() { ... }
...
void f()
{
  enum e_types c = classify();
  assert(c >= 0 && c < 4);
```

Figure 2. Enforcing interfaces with an interval analysis

In this example a larger piece of software has been analyzed by an interval analysis. This analysis computes the intervals of values that an integer variable holds at each location. By executing the assertions of the analysis results the transformed program virtually performs a dynamic range check for the used range of the value domain. These restricted value ranges often reflect implicit conventions introduced by the programmers and can be visualized with behaviour-carrying code. The programmers can review the generated assertions and use them to freeze certain aspects of the software against undesirable future modifications.

4.2 For the authors of analyzers: Automatic testing

In our daily work of implementing static program analyses the topic of testing is an important issue. Realistically it is not possible to create a bug-free implementation of a program analysis without thorough testing, since it is very easy to miss certain corner-cases. Bugs in the analyzer can have two reasons: Either they come from an incomplete or even wrong specification or they can be traced to errors in the implementation. In both cases, they will surface as an analysis result that can not be validated against the program at run-time. By applying the behaviour-carrying code transformation to an analyzed program, the resulting program can be used as a test for the analysis: If an assertion fires, the analysis result that was tested for did not match the reality. This method of simply executing the transformed program, however, does have the drawback of uncertain path coverage. Typical test suite programs often lack proper entry points/*main*-functions or are otherwise incomplete. To overcome this issue, one solution is to also generate stubs for these missing functions.

In the SATIrE implementation, for example, we create an artificial main function that calls the other functions in the module that are to be tested. Such a generated main function of course needs to adhere the calling conventions of the other functions in the module. For parameterless functions, this is easily achieved, otherwise it is necessary to fake input values depending on the function signature. A naïve approach would be to call every function in the module. To improve the performance when testing multiple combinations of input values, it pays off to take the call graph into account to reduce the overall number of function calls. By calculating the transitive closure of the call graph, a minimal set of function "clusters" can be identified. One call to the first node of each cluster guarantees that every function is reachable from the main function.

If the execution speed is circumstantial, there is also the option to use a formal verification engine to prove the assertions. (Bounded) software model checkers such as Blast [BHJM07] and CBMC [CKL04] fill this niche. In our experience CBMC seems to be more suited for this kind of programs as it was consistently faster in our experiments. Using a model checker, we can get a verification of the analysis result. If the model checker produces a counter-example we know that either the analysis specification, analysis implementation or the model checker is erroneous. It must be noted that the runtime of model checkers can be quite unpredictable. In our experiments the model checker failed to terminate within reasonable time or space requirements for a substantial number of test cases. This is because, in the worst case, the model checker has to examine every vector of the state space of the program.

As an interesting twist, the combination of model checking and automatic assertion insertion can also be used in the opposite direction. Since the model checker can find out whether an analysis result is valid, we can turn the tables by repeatedly making an *informed guess* until the model checker can prove it to be correct. Using search strategies like binary widening and binary tightening we can "analyze" information where cheaper methods fail. For a detailed experience report the reader is referred to [PKK+09].

From our experience we can conclude that a combination of test executions and model checking (with sensitive time- and memory limitations) is a valuable means to aid the development of static program analyzers. It helped us to pinpoint problems introduced by recent changes in the interprocedural control flow graph construction that otherwise would probably have gone unnoticed until much later.

5 Implementation

We implemented the BCC transformer for the SATIrE source-to-source analyzers. The analyzers generated by SATIrE operate on an interprocedural control flow graph built from a C++ abstract syntax tree (AST). The analysis results are stored in attributes of the AST. The transformer implementation was written in Prolog using the TERMITE[1] library. This library contains abstractions, traversals and query predicates that facilitate operating on the external term representation that is exported by SATIrE analyzers.

Figure 3 shows the relevant portion of a BCC transformer for a loop bound analysis. A loop bound analysis is concerned with finding upper bounds for the trip count of loop constructs. The transformer works by introducing a new unsigned integer variable that is initialized to zero before the loop is entered. The new variable is incremented by one for each iteration of the loop body. After the loop, an assertion is inserted stating that the counter variable is smaller or equal than the loop bound. An example of a program transformed this way is shown in Figure 4. The transformer reads the loop bound attribute from the loop body, generates the counter and places the loop inside of a new compound statement[2] that serves as scope for the counter variable. The term representation also contains bookkeeping information about source file location of every node, which was replaced by "..." for improved readability in the listing.

6 Extensions: Context sensitivity

In this section we investigate how to take context information tethered to call strings into account, which leads to more precise interprocedural analyses [SP81]. Intuitively, call strings allow keeping call contexts up to a predefined length n (= length of the call string) separate. As an illustrating example, the result of an interprocedural interval analysis with call string length 1 and its context-insensitive counterpart is given in Figure 5. To account for context-sensitivity in behaviour-carrying code, several options are available:

1. Introducing an extra *call-string*-argument to every function call that keeps track of the current context. This has a slight impact on performance and dynamic memory footprint caused by the additional bookkeeping.
2. Generating explicit copies of the deepest n functions in the expanded call graph. While this transformation is likely to increase execution speed, it has the drawback of increasing code size significantly.

Both methods involve a major refactoring of the original program. While a less invasive method would seem preferable, it has to be noted that the modified program also carries an increased potential for subsequent optimizations that is comparable to that of inlining.

[1] Termite Web Page: `http://www.complang.tuwien.ac.at/adrian/termite/`

[2] For historic reasons the compound statement is misleadingly called "basic block" by the underlying ROSE compiler.

```
assertions(..., Statement, AssertedStatement) :-
  Statement = while_stmt(Test, basic_block(Stmts, ...), ...),
  get_annot(Stmts, wcet_trusted_loopbound(N), _),

  counter_decl('__bound', ..., CounterDecl),
  counter_inc('__bound', ..., Count),
  counter_assert('__bound', N, ..., CounterAssert),

  AssertedStatement =
    basic_block([CounterDecl,
                 while_stmt(Test, basic_block([Count|Stmts], ...),
                            ...),
                 CounterAssert], ...).
```

Figure 3. Source-to-source transformer for loop bounds

```
int binary_search(int x) {
  int fvalue, mid, up, low = 0, up = 14;
  fvalue = -1; /* all data points are positive */
  {
    unsigned int __bound = 0;
    while(low <= up){
      ++__bound;
      mid = low + up >> 1;
      if (data[mid].key == x) { /* found */
        up = low - 1;
        fvalue = data[mid].value;
      }
      else if (data[mid].key > x)
        up = mid - 1;
      else low = mid + 1;
    }
    assert(__bound <= 7);
  }
  return fvalue;
}
```

Figure 4. Generated assertions for loop bounds

```
int g(int x) {
// pre info: Context(0,0):[(TOP,TOP):x->(0,0)]
//           Context(0,1):[(TOP,TOP):x->(42,42)]
//           Merged:[(TOP,TOP):x->(0,42)]
  return x + x;
// post info: Context(0,0):[(TOP,TOP):x->(0,0),$tmpvar$retvar->(0,0)]
//            Context(0,1):[(TOP,TOP):x->(42,42),$tmpvar$retvar->(84,84)]
//            Merged:[(TOP,TOP):x->(0,42),$tmpvar$retvar->(0,84)]
}

int f(int x) {
  return g(0);
}

int h(int x) {
  return g(42);
}
```

Figure 5. Calling contexts and merged interval information

7 Conclusions

With behaviour-carrying code we presented a portable way to make the results of static program analyses available for a wide range of applications. While the testing of analyzers will be of interest particularly for compiler developers, the interoperability perspectives could prove most useful and valuable for other users as well. As we have seen in the loop bound example, the transformation is not obvious for all types of analyses. However, what might look as a shortcoming of the approach can also be regarded as a chance: By compiling analysis results into information that can be validated easily (and most important: automatically and locally!), compilers can suddenly profit from new analyses without having to re-implement them. This way we hope to increase the collaboration between stand-alone static analyzers and compiler systems in the future.

The validation of static program analyzers provided the original inspiration for the presented transformation and was already successfully implemented in the static analysis framework SATIrE.[3]

Acknowledgements The authors would like to thank Gergő Bárány for many helpful comments on earlier versions of this article.

References

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools.* Pearson Education, second edition, 2007.

[3] SATIrE web page: `http://www.complang.tuwien.ac.at/satire/`

[BHJM07] Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, October 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, CA, 1997.

[Nec97] George C. Necula. Proof-carrying code. In *Conf. Rec. 24th Annual Symp. on Principles of Prog. Lang.* (*POPL'97*), pages 106 – 119. ACM, NY, 1997.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, 1998.

[PKK+09] Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan. From trusted annotations to verified knowledge. In *On-Site Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009)*, Dublin, 2009.

[Sch08] Markus Schordan. Source-to-source analysis with satire - an example revisited. In Florian Martin, Hanne Riis Nielson, Claudio Riva, and Markus Schordan, editors, *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.