Source-To-Source Analysis with SATIrE - an Example Revisited

Markus Schordan

Institute of Computer Languages
Vienna University of Technology, Austria
markus@complang.tuwien.ac.at

Abstract. Source-to-source analysis aims at supporting the reuse of analysis results similar to code reuse. The reuse of program code is a common technique which attempts to save time and costs by reducing redundant work. We want to avoid re-analyzing parts of a software system, such as library code. In the ideal case the analysis results are directly associated with the program itself. Source-to-source analysis supports this through program annotations. Further more, to get the best out of available software analysis tools, we aim at enabling the combination of the analysis results of different tools. In order to allow this, tools must be able to process another tool's analysis results. This enables numerous applications such as automatic annotation of interfaces, testing of analyses by checking the results of an analysis against provided annotations, domain aware analysis by utilizing domain-specific program annotations, and making analysis results persistent as annotations in source code.
The design of the *Static Analysis Tool Integration Engine* (SATIrE [6]) allows to map source code annotations to its intermediate program representation as well as generating source code annotations from analysis results that are attached to the intermediate representation. The technical challenges are the design of the analysis information annotation language, the bidirectional propagation of the analysis information through different phases of the internal translation processes, and the combination of the different analyses through the plug-in mechanism. In its current version SATIrE targets C/C++ programs. In this paper we present the approach of source-to-source analysis and show in a detailed example analysis how we support this approach in SATIrE.

1 Introduction

Source-to-source analysis enables the interoperability of analysis tools at the source level. The essential idea is that analysis results are represented as annotations of source code such that they can be reused in subsequent analyses of the source code. The annotations can be both generated and read in, also allowing manual annotation of source code with domain specific knowledge. Hence, a source-to-source analyzer has beside the usual Front End and Back End for the source language, also an Annotation Front End as well as an Annotation Back End.

Dagstuhl Seminar Proceedings 08161
Scalable Program Analysis
http://drops.dagstuhl.de/opus/volltexte/2008/1569

An example is the analysis of a library where for each function it is determined whether it has side effects. This information is then made available as annotation of the interface where the function is accessible. Source-to-source analysis starts to show its full potential if we not only consider properties at the function level, but also flow-sensitive and context-sensitive information. In order to annotate programs at this level of detail, we must reason about associating information with a certain position in a program dependent on the flow and context of the program. This is complicated by the fact that we need to map information obtained from the annotated source code through multiple levels of intermediate representations. The multiple levels of intermediate representation become necessary with the separation of different kinds of analysis information, such as control flow and data flow information, as well as calling contexts for inter-procedural analysis.

To enable tool interoperability at the source level we demand the following properties for source-to-source analysis:

1. Analysis information is associated with locations in the source code.
2. No internal information of an analyzer is exposed in the analysis results.
3. Analysis information annotations can be both input as well as output.

That the analysis information is associated with locations in the original source code is the basis for presenting analysis information such that developers can use the results with appropriate tools for better understanding, verifying, and debugging their programs. To not expose internal information of the analyzer, is a requirement for tool interoperability and an essential property of source-to-source analysis. Otherwise it would mean to simply dump the internal structure and the computed analysis information - instead we want to ensure that the computed analysis information is made consistent with the original source code and analysis results are presented as annotations of the original program. That annotations can be input to the analyzer allows to reuse analysis results computed in a previous run of the analyzer, or to also read in manual annotations. This can be useful for providing test cases for the analyzer implementation or provide information about parts of the program source that are not accessible to the analyzer.

A source-to-source analyzer has in common with a compiler that it normalizes or lowers program code before performing a program analysis. The reason is that the reduced number of different language constructs allows for a more compact analysis specification because only a reasonably small number of different language constructs must be explicitly addressed in the analysis. Compilers usually continue to generate machine code either for a concrete hardware platform or for a virtual machine. In contrast, a source-to-source analyzer propagates analysis information back to the original source code representation through all intermediate levels and generates the analysis information associated with the original source code.

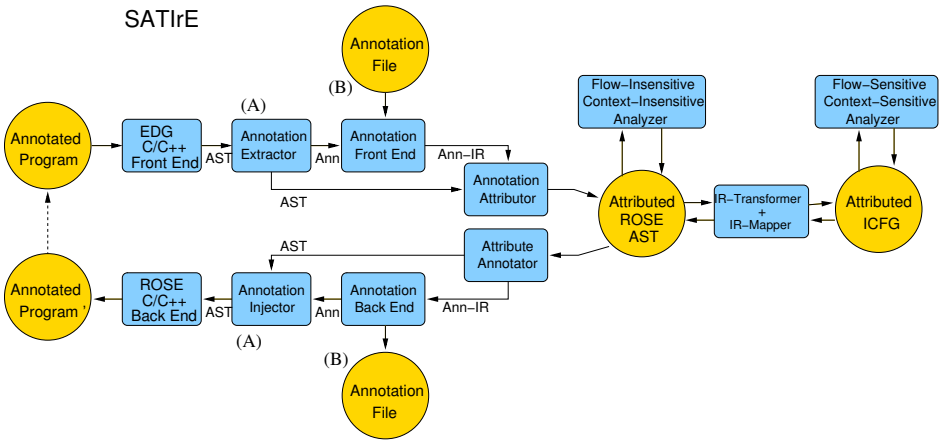

Fig. 1. SATIrE Source-To-Source Architecture

2 Source-To-Source Analysis Architecture

In Fig. 1 the components of the SATIrE source-to-source analyzer architecture are shown. The analysis information can be represented as annotations of the source file, marked (A), or in a separate annotation file, marked (B). Once an analysis has been performed we can either generate an annotation file, marked (B) or add analysis results as annotations to the input program, which is generated as new source code by the Back End.

SATIrE integrates the LLNL-ROSE infrastructure [7] and the Program Analyzer Generator (PAG [5]) from AbsInt. From the ROSE infrastructure we use its connection with the EDG C++ Front End, its intermediate representation, the ROSE-AST, and the ROSE C++ Back End. PAG allows to specify abstract interpreters to perform a flow-sensitive and context-sensitive analysis. For handling annotations and analysis results, the following components are part of the SATIrE architecture

EDG C/C++ Front End. We use the C++ Front End of the Edison Design Group (EDG [3]) for parsing C/C++ programs. The EDG-ROSE connection (not shown) translates the EDG-IR to an object-oriented AST, suitable for source-to-source program transformations.

Annotation Extractor. This SATIrE component extracts annotations from the AST and represents them in the same syntax format as they are represented in an alternative annotation file. We annotate C++ programs with pragma statements. The strings of the pragma statements are extracted from the AST and passed as a list to the Annotation Front End.

Annotation Front End. Parses annotations, either provided in a separate annotation file or extracted from source code. It generates an Annotation-IR (Ann-IR).

Annotation Attributor. Takes as input the Annotation-IR and the AST and attaches the Annotation-IR elements as attributes to the AST. The identification of the corresponding locations is obtained from the annotation and its associated label.

Flow-Insensitive Context-Insensitive Analyzer. Performs analyses on the attributed ROSE AST and attaches analysis results as attributes to the AST.

IR-Transformer and IR-Mapper. The IR-Transformer translates an AST into an inter-procedural control flow graph (ICFG) and normalizes or lowers the statements. The IR-Mapper maps analysis results (either obtained from annotations or previous analyses) from the ROSE-AST to the ICFG. The IR-Mapper also performs the mapping of analysis results from the ICFG to the AST.

Flow-Sensitive Context-Sensitive Analyzer. For specifying flow-sensitive and context-sensitive analyses we have integrated PAG into SATIrE. The ICFG is designed such that PAG can operate on it and perform an abstract interpretation of a C/C++ program. The analysis results are attached as attributes to the ICFG and mapped back by the IR-Mapper.

Attribute Annotator. Takes as input the attributed AST, where the attributes represent analysis results, and translates the attributes into Annotation-IR elements.

Annotation Back End. Generates an annotation file or a list of annotation strings as input to the Annotation Injector.

Annotation Injector. Adds annotation strings as pragma statements to the AST.

ROSE C/C++ Back End. Generates C/C++ source code from an AST. If annotations have been added to the AST as pragmas, the Back End generates an annotated C/C++ source code.

The two approaches, (A) adding annotations directly to a program, or (B) using a separate annotation file, are designed to be equivalent in expressiveness, and the method most suitable to exchange analysis information with a given tool should be chosen.

3 Mapping Analysis Results

After an analysis the analysis result is propagated back through the intermediate levels to the source code level. This is performed by the analysis results IR-Mapper, depicted in Fig. 1. The mapping is defined in both directions, to map analysis information from the AST to the ICFG and vice versa. This mapping must take the semantics of the performed analysis into account, to ensure that the analysis information is consistent with each IR. The example shown in Section 5 provides details on how this mapping is currently performed in SATIrE for a reaching definition analysis.

4 Analysis Results Annotation Language (ARAL)

The analysis results annotation language is designed to be suitable for annotating flow-sensitive and context-sensitive analysis results. It allows to represent computed data of analysis tools, but without putting restrictions on the semantics of an analysis. Clearly, the annotations allow to represent analysis result data, but the semantics of the data are defined for each analysis separately. For example, the language is general enough to represent analysis information specified in PAG's [5] DATLA language. Since PAG is integrated into SATIrE this is a requirement for reusing any analysis information computed with a PAG generated analyzer. ARAL consists of sets, lists, tuples, maps, and some primitive data types. Additionally each analysis information has an Analysis Identifier. This identifier allows to associate semantics with the analysis information.
An ARAL analysis information annotation consists of:

Analysis Identifier. Allows to identify the analysis information and associate the data with semantics.

Location. Each analysis information is associated with a program location and its corresponding program fragment. The program location is represented by an optional Location Reference and a unique label. The label is represented by a unique number and a mapping of labels is maintained by the IR-Mapper (see Section 2) between the AST and the ICFG representation. The Location Reference allows to specify with which part of a program fragment an annotation is associated. This allows to annotate multiple subexpressions of an expression without breaking up the expression. An example is param(N) for specifying an annotation of the Nth formal parameter of a function, or 'entry' and 'exit' for denoting that an annotation is the entry or exit information of a function (see Fig 3).

Flow-Specifier. A Flow-Specifier allows to define whether an analysis information is a pre or post information in a flow-sensitive information. If the information is flow-insensitive it is denoted as '-'.

List of Data Elements. A data element consists of

- **Context Identifier.** For a context-sensitive analysis each context is represented in the annotations by an identifier which allows to keep context sensitive information separate. For a context-insensitive analysis the context identifier is the same for all analysis results computed for a program or function.
- **Analysis Data.** The analysis data represents the analysis information computed at a location in a program. A location can be associated with a function, a statement, an expression, or with a scope.

Analysis data can consist of a set, list, tuple, map, string, number, or specific program information: numeric VariableIds, ExpressionIds, StatementIds, FunctionIds and Labels. Two special symbols exist for the top and bottom element of lattices. Sets, lists, and maps can only contain data elements of the same type, whereas tuples can contain elements of different type. In a data element the types of data can be arbitrarily nested and combined.

5 Example

The generation of annotations for a given program can be done in two different forms. Either the annotations are added to a given program or in a separate file. Here we show an example with annotations added to the input program. Since the input program is C++ source code, we represent annotations in a specific syntax, the ARAL language (see Section 4), in pragma statements.
In Fig. 2 and Fig. 3 the result of a context-sensitive reaching definition analysis is shown. The annotation information consists of the Analysis Identifier, RD, the Flow Specifiers pre and post, the Context Identifiers Cα where α is a unique number for each calling context, and the Analysis Data consisting of a set of pairs. The pairs represent the reaching definitions, with the first item being the defined variable, and the second one being the label of the program location of the assignment of the variable. The analysis is a forward may analysis. The labels of the analysis data represent locations in the source program. The locations are denoted by a unique number, a label, in the annotations as well, and sometimes extended with a category identifier, such as param(N) for the Nth formal parameter of a function. Pre/post denotes the pre and post information computed for the respective statement.
We next describe how the RD analysis results are generated as source code annotations for use by other analysis tools or for reuse of SATIrE. We then show how we can read in RD analysis results and map them from the AST to the ICFG such that a subsequent analysis operating on the ICFG can reuse this results. Hence, we can support separate analysis and accomplish whole program analysis.

5.1 Source Code Annotations as Output

After the RD analysis has been performed on the ICFG the analysis results are mapped from the ICFG to the AST and added as annotations. Then the Back End is invoked and the source code shown in Fig. 2 and Fig. 3 is generated. We have added some additional information, being crossed out, to also discuss what is not shown in the generated source code although it is computed by the analyzer.
The ICFG consists of a number of different types of nodes which do not exist in the AST. These nodes are necessary to a) separate the control flow from the AST, and b) provide specific source and target nodes for inter-procedural edges in the ICFG. In Fig 5 the respective ICFG for the program shown in Fig. 2 and 3 is shown. The mapping performed by the IR-Mapper is illustrated by marking in dark grey the nodes of which pre and post information is mapped to the AST (at the source level), and in light grey the nodes for which either pre or post information only is mapped to the AST. For white nodes no analysis information is mapped to the AST. For all mapped nodes the labels of the nodes that are shown in 5 are the same as in Fig. 2 and 3. The mapping of labels is performed such that the reaching definition information does not need to be altered, except that temporary variables that only exist in the ICFG are

```
int d; // global variable declaration
int inc(int); // function declaration

#pragma ARAL RD entry:2 pre C1:{(d,-1)}
int main()
{
  #pragma ARAL RD 30 pre C1:{(d,-1)}
  int a;
  #pragma ARAL RD 30 post C1:{(d,-1),(a,-1)}
  #pragma ARAL RD 29 pre C1:{(d,-1),(a,-1)}
  int b;
  #pragma ARAL RD 29 post C1:{(d,-1),(b,-1),(a,-1)}
  #pragma ARAL RD 28 pre C1:{(d,-1),(b,-1),(a,-1)}
  int c;
  #pragma ARAL RD 28 post C1:{(d,-1),(b,-1),(a,-1),(c,-1)}
  #pragma ARAL RD 27 pre C1:{(d,-1),(b,-1),(a,-1),(c,-1)}
  a = 3;
  #pragma ARAL RD 27 post C1:{(d,-1),(b,-1),(a,27),(c,-1)}
  #pragma ARAL RD 26 pre C1:{(d,-1),(b,-1),(a,27),(c,-1)}
  b = a;
  #pragma ARAL RD 26 post C1:{(d,-1),(b,24),(a,27),(c,-1)}
  #pragma ARAL RD 25 pre C1:{(d,-1),(b,-1),(a,27),(c,-1)}
  d = 1;
  #pragma ARAL RD 25 post C1:{(a,27),(c,-1),(d,25),(b,26)}
  #pragma ARAL RD 11 pre C1:{(a,15),(c,-1),(d,6),(d,25),(b,26),(a,27),(b,16),(c,12)}
  while(a < 10) {
    #pragma ARAL RD 14 pre C1:{(a,15),(c,-1),(d,6),(d,25),(b,26),(a,27),(b,16),(c,12)}
    if (a < b) {
      #pragma ARAL RD 15 pre C1:{(a,15),(c,-1),(d,6),(d,25),(b,26),(a,27),(b,16),(c,12)}
      a = (a + 1);
      #pragma ARAL RD 15 post C1:{(a,15),(b,16),(c,-1),(d,6),(d,25),(c,12),(b,26)}
    }
    else {
      #pragma ARAL RD 21 pre C1: {(a,15),(c,-1),(d,6),(d,25),(b,26),(a,27),(b,16),(c,12)}
                                 ->C2,C3
      b = (inc(b) + inc(b));
      #pragma ARAL RD 16 post C1: {(a,27),(b,16),(a,15),(d,6),(c,-1),(c,12)} <-C2,C3
    }
    #pragma ARAL RD 13 post C1:{(a,15),(d,6),(c,-1),(d,25),(b,26),(a,27),(b,16),(c,12)}
    #pragma ARAL RD 12 pre C1:{(a,15),(c,-1),(d,6),(d,25),(b,26),(a,27),(b,16),(c,12)}
    c = (a + b);
    #pragma ARAL RD 12 post C1:{(a,27),(a,15),(b,16),(d,6),(c,12),(d,25),(b,26)}
  }
  #pragma ARAL RD 10 post C1:{(a,15),(c,-1),(d,6),(d,25),(b,26),(a,27),(b,16),(c,12)}
  #pragma ARAL RD 9 pre C1:{(a,15),(c,-1),(d,6),(d,25),(b,26),(a,27),(b,16),(c,12)}
  d = d + c;
  #pragma ARAL RD 9 post C1:{(a,27),(a,15),(b,16),(c,-1),(c,12),(d,9),(b,26)}
  #pragma ARAL RD 8 pre C1:{(a,27),(a,15),(b,16),(c,-1),(c,12),(d,9),(b,26)}
  return 0;
  #pragma ARAL RD 8 post C1:{(a,15),(c,-1),(d,9),(b,26),(a,27),~~($retvar,8)~~,(b,16),(c,12)}
}
#pragma ARAL RD exit:3 post C1:{~~($retvar,8)~~}
```

Fig. 2. Example: Annotated function main with context-sensitive reaching definition information. Crossed out items are related to temporary variables of the IR and not included in the source code annotation. The function inc is shown in Fig. 3.

```
#pragma ARAL RD entry:0 pre C2:{(d,6),(d,25),($arg-0,21)}, C3:{(d,6),,($arg-0,17)}
#pragma ARAL RD param(1):4 pre C2:{(d,6),(d,25),($arg-0,21)}, C3:{(d,6),,($arg-0,17)}
#pragma ARAL RD param(1):4 post C2:{(x,4),(d,6),(d,25)}, C3:{(x,4),(d,6),}
int inc(int x)
{
  #pragma ARAL RD 6 pre C2:{(x,4),(d,6),(d,25)}, C3:{(x,4),(d,6),}
  d = d + 1;
  #pragma ARAL RD 6 post C2:{(x,4),(d,6),}, C3:{(x,4),(d,6),
  #pragma ARAL RD 5 pre C2:{(x,4),(d,6),}, C3:{(x,4),(d,6),
  return x + 1;
  #pragma ARAL RD 5 post C2:{(x,4),(d,6),($retvar,5)}, C3:{(x,4),(d,6),,($retvar,5)},

}
#pragma ARAL RD exit:1 post C2:{(d,6),($retvar,5)}, C3:{(d,6),,($retvar,5)},
```

Fig. 3. Example: Annotated function inc with context-sensitive reaching definition information. Crossed out items are related to temporary variables of the IR and not included in the source code annotation.

eliminated. The information corresponding to temporary variables introduced in the ICFG is crossed out in Fig. 2 and 3. For the mapping used here and the RD analysis it is guaranteed that the labels in the generated annotations are consistent at the source code level. An important aspect is that we do not generate all information computed at the ICFG level, because the information about temporary variables would mean that we generate internal information in the annotations - and this would violate our Property 2 in the required properties of a source-to-source analyzer (see Section 1).

In Fig. 4 the ICFG nodes representing the statement b=inc(b)+inc(b) are shown. These are the nodes 21,23,24,22,17,19,20,18,16. Only two of these nodes, 21 and 16, are marked as light grey in Fig. 5, meaning that only information of those two nodes is mapped to the AST. Here it is the pre-info of node 21 and the post-info of node 16. Both are mapped to positions in the source code, denoted with the same label[1]. The other nodes of this statement are computed by the analyzer and necessary to make the control flow explicit and introduce specific nodes, such as the call node, 23, the return-from node, 24, for providing source and target nodes for inter-procedural control flow. The other assignment nodes, for handling parameter binding and return values, are introduced by the code normalization of the IR-Transformer.

ARAL also allows to specify analysis information for locations inside expressions. For example, to annotate the analysis information before and after the binding of the actual parameters for the second call in the expression b=inc(b)+inc(b) we could write

[1] Introducing a level of indirection the labels can be mapped to other numbers at the source code level, but to ease the discussion we use the same label numbers at the ICFG and AST if the same analysis information is mapped.

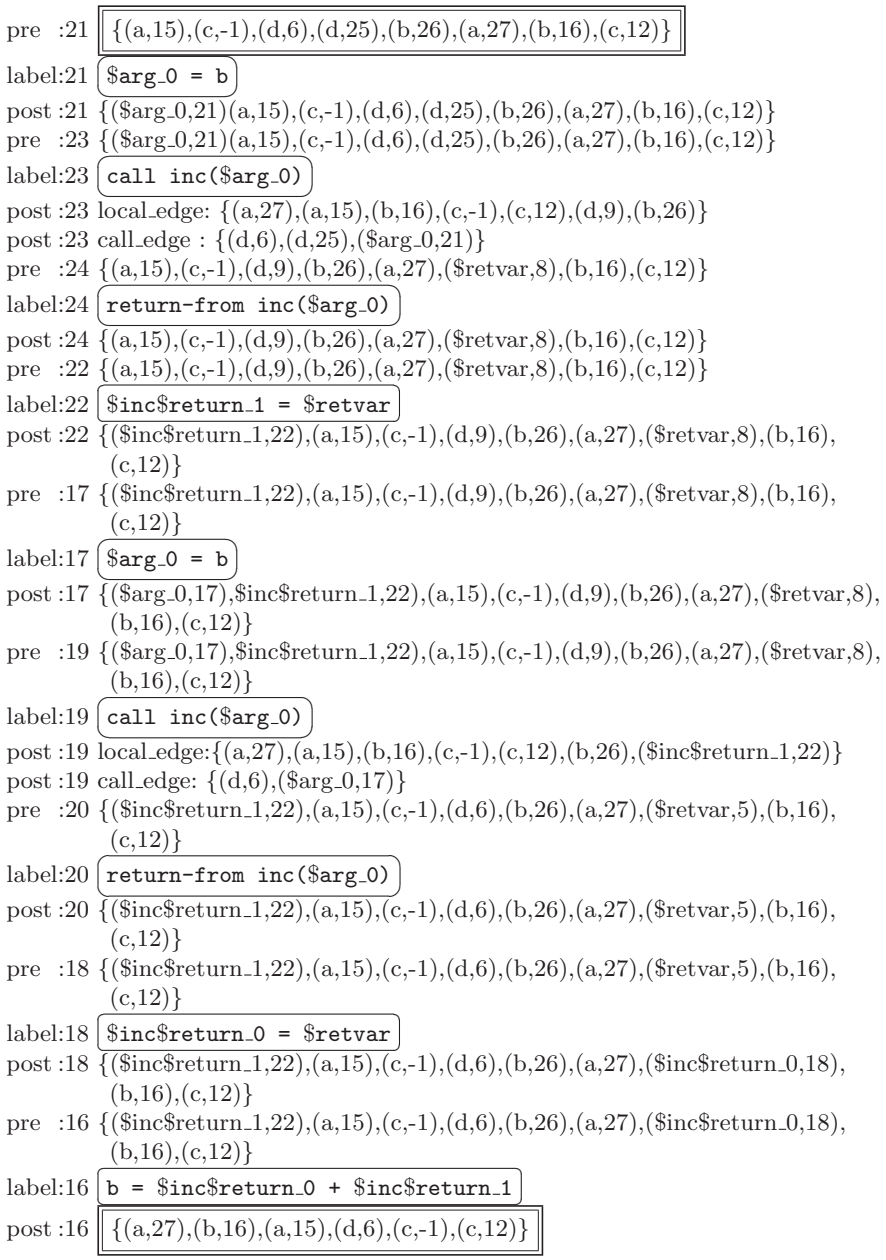

Fig. 4. Example: ICFG fragment representing the statement b=inc(b)+inc(b) in normalized form. The double-framed boxes show which information is mapped to source code locations (see Fig. 2). Pre/post denotes the flow-sensitive analysis information. The shown framed nodes are connected in the listed order.

```
#pragma ARAL RD call(2):param(1):17 pre {($inc$return_1,22),(a,15),(c,-1),(d,9),(b,26),
        (a,27),($retvar,8),(b,16),(c,12)}
#pragma ARAL RD call(2):param(1):17 post {($arg_0,17),$inc$return_1,22),(a,15),(c,-1),
        (d,9),(b,26),(a,27),($retvar,8),(b,16),(c,12)}
b = inc(b) + inc(b);
```

This annotated information is the same as for label 17 in Fig. 4 but without temporary variables. In the complete example in Fig. 2 this annotation is not added because in the example we only annotate analysis information at the statement level and the binding of formal parameters.

5.2 Source Code Annotations as Input

Reading and Mapping source code annotations through all intermediate levels is complicated by the fact that the annotations do not contain all information that is computed on the ICFG level. Therefore this information needs to be either generated by a sophisticated mapping by taking the semantics of the analysis into account, or by rerunning the analysis after proper initialization with all available information.

Our label mapping allows to rerun the RD analysis and compute the RD information for temporary variables without modification of the annotation information. Clearly, for our example the IR-Mapper maps the information associated with the labels in the source code (Fig. 2 and 3) to nodes being marked light grey or dark grey in Fig. 5 (labels shown with identical numbers). Thus it is the inverse mapping operation as discussed in the previous section w.r.t. to marked nodes. To establish the analysis information for the non-marked nodes is not straight-forward because it cannot be obtained by directly mapping analysis information. For our example in Fig. 5 this means that the analysis information for the non-marked nodes must be computed according to the semantics of the transfer functions of these nodes. However, the calculation required is limited by the fact that the result of the analysis for node 21 is already available. The local temporary variables, $retvar, increturn_0 and increturn_1 are passed on the local edge of the call node and only exist until node 18 and 16, respectively. The scope of these temporary variables ends there and they are eliminated from the analysis information set. The temporary variable $arg_0, used for handling parameter binding, only exists from its initialization, node 21, to its use in the respective called function inc, where it is assigned to the formal parameter, node 4p. The temporary variable at node 17 is a temporary variable with the same name but separate scope. Thus, for these language constructs the recomputation of the temporary variables does not require another fixpoint computation, but only a constant number of computations of transfer functions. Short circuit evaluation of boolean expressions is another case where temporary variables are introduced, not shown here, but similar rules apply there.

Thus, for the RD analysis, we can read in analysis results from annotations and establish the complete information at every IR level, the AST and the ICFG, without another fixpoint search, by computing sequences of transfer functions involving temporary variables of our IR only. To establish the information at the

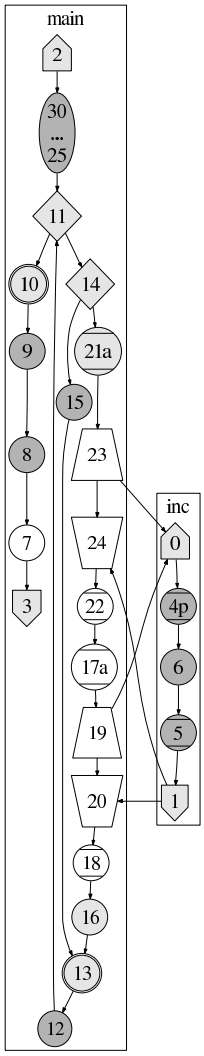

Fig. 5. ICFG of the example program shown in Fig. 2 and Fig. 3. The grey nodes are directly mapped to locations at the source code level and represented as label annotations. From dark grey nodes both pre and post information is mapped, from light grey nodes only either one, pre or post information is mapped. A node with a house-shape represents a function entry node, with an inverse house-shape a function exit node. Trapezoid nodes represent call and return nodes. Condition-s/branches are represented by diamond nodes. A double-circle node represents a join node. Circle nodes with two bars represent nodes with assignments to temporary variables. Node labels postfixed with character 'a' represent actual parameter assignment, and with 'p' formal parameter assignment. The other temporary variable nodes represent function return value handling.

AST is important to allow fast flow-insensitive context-insensitive analyses (see Fig. 1), and for combining flow-sensitive context-sensitive analyses it is necessary to establish consistent analysis information at the ICFG level.

6 Related Work

Harrold and Rothermel present a technique for separate analysis of modules [4]. The work focuses on one particular analysis, inter-procedural may alias analysis, but the design of the analyzer is general and similar to our setting. For inter-procedural analysis an inter procedural control flow graph (ICFG) is created. The separation in control flow and intermediate representation of statements and expressions is the same as in our approach. The analysis is a modular analysis, meaning that a module is a set of interacting procedures or a single procedure that has a single entry point. The approach allows to reuse the analysis results after analyzing a module and thus, is applicable to large scale software and real world applications. In our approach we can add analysis results as annotations to source-code, allowing to reuse analysis results in a subsequent analysis step. This can either be done on the IR-level, or the annotated source code is read in again.

An approach for user-defined checks that are performed by a compiler is presented in [8]. User-defined checks may increase the confidence of a programmer with respect to his code, especially if used on a continuous basis during development. The checks that can be expressed can refer simultaneously to syntax, semantics, control flow, and data flow. The tool Condate allows more semantic-enabled and user-centric compilers, obtained by fusing together compilation with other powerful analyzers. By using SATIrE such tools can be built by using the annotation mechanism and checking manual annotations with an appropriate analysis.

For optimizing compilers the automatic generation of data flow analyses and optimizations out of concise specifications has been a trend for several years. The systems of [1,2] concentrate on "classical" inter-procedural optimizations, whereas the system of [9] is particularly well suited for local transformations based on data dependency information. We integrated PAG [5] because it is a tool that allows to generate analyzers from specifications for similar analysis problems.

7 Conclusion

Source-To-Source analysis allows to generate analysis results as source code annotations, but also to read annotations in and map them to the internal representation. The corner stones of source-to-source analysis can be summarized as follows:

- Analysis information is represented as annotations.
  - Annotations can be read and mapped to IR (Front End)

  - Annotations can be generated from IR (Back End)
- Annotations are associated with source code locations.

Source-to-source analysis aims at making analysis results independent of a certain tool and permits exchange of analysis results between different analysis tools. It also supports whole-program analysis by making analysis results persistent as source code annotations. We demonstrated with an example of the classic reaching definitions analysis, how we can perform source-to-source analysis with SATIrE.
Currently we are investigating how to determine a class of analyses for which the presented mapping of analysis information between source code and IR is sufficient such that analysis information can be computed on all additionally introduced IR nodes without performing a fixpoint iteration.

Acknowledgements. This work has been partially supported by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/), and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

References

1. U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proceedings of the 6th International Conference on Compiler Construction* (*CC'96*) (*Linköping, Sweden*), Lecture Notes in Computer Science, vol. 1060, pages 121 – 135. Springer-Verlag, Heidelberg, Germany, 1996.
2. U. Aßmann. On edge addition rewrite systems and their relevance to program analysis. In *Proceedings of the 5th International Workshop on Graph Grammars and Their Application to Computer Science* (*GGTA'94*) (*Williamsburg*), Lecture Notes in Computer Science, vol. 1073, pages 321 – 335. Springer-Verlag, Heidelberg, Germany, 1996.
3. Edison Design Group. http://www.edg.com.
4. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Softw. Eng.*, 22(7):442–460, 1996.
5. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
6. SATIrE. http://www.complang.tuwien.ac.at/markus/satire. Static Analysis Tool Integration Engine.
7. M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In L. Böszörményi and P. Schojer, editors, *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.
8. N. Volanschi. Condate: a proto-language at the confluence between checking and compiling. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 225–236, New York, NY, USA, 2006. ACM Press.
9. D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053 – 1084, 1997.