Linking Analysis and Transformation Tools with Source-based Mappings

Magiel Bruntink
CWI, P.O Box 94079
1098 SJ Amsterdam, The Netherlands
Magiel.Bruntink@cwi.nl

Abstract

This paper discusses an approach to linking separate analysis and transformation tools, such that analysis results can be used to guide transformations. Our approach consists of two phases. First, the analysis tool maps its results to relevant locations in the source code. Second, a mapping in the reverse direction is performed: the analysis results expressed as source positions and data are mapped to the abstractions used in the transformation tool. We discuss a prototype implementation of this approach in detail, and present the results of two applications.

1. Introduction

There exists a vast collection of source code analysis and transformation tools. Most of these tools specialize in either analysis or transformation, and rarely a tool is suitable for both tasks. Ironically, most non-trivial transformation tasks require deep analysis. Migrating legacy software to recent technology, such as AOP, is but one example [5].

Combining tools is the obvious solution to this functionality schism. However, tool combination introduces the issue of tool interoperability, and despite the ample attention it has received from the research community, it still remains a largely open problem [8]. Previous work in this area has focused on solving low-level compatibility issues, resulting in many proposals for generic data formats, and communication protocols [12, 23, 3, 9, 13]. These technologies have proved to be useful in several successful tool collaborations such as the ASF+SDF Meta Environment [4]. Another key feature of tools like the ASF+SDF Meta Environment is that they operate on an abstract representation (i.e., ASTs) that is shared by all of their components. A common term for such an abstract representation is *schema*, which we will use throughout this paper.

A remaining challenge consists of coping with differences between schemas (of the source code) employed by the various tools. For instance, an analysis tool such as Grammatech's CodeSurfer [2] revolves around program dependence graphs (PDGs). In contrast, transformation tools such as ASF+SDF [4] operate primairily on abstract syntax trees (ASTs). To leverage analysis results expressed in the PDG domain (i.e., CodeSurfer), it is first required to map the analysis results to the AST domain (i.e., ASF+SDF). Clearly, creating such a mapping (or *bridge* [8]) is a non-trivial task, requiring deep understanding of the schemas used at both ends. Even if both tools are targeted at the same language, the use of different grammars, language dialects, and source correspondences complicate this task enormously, especially since it is often hard to change those features of a tool.

In this paper we discuss an approach to create *source-based mappings* between tools using different schemas. A source-based mapping consists of pairs of a source code area and facts relevant at that area. The perspective taken for this discussion consists of two tools working together on the same source code; one tool performs the analysis required for the transformations performed by the other tool. We will define when a source-based mapping is *strict* and *safe*, given the relevant abstractions in the analysis and transformation tools, and a body of source code. A strictly safe source-based mapping guarantees that analysis results are mapped to the desired abstraction in the transformation tool. Furthermore, we show how a source-based mapping compares to mappings created using higher-level schemas.

The paper is organized as follows. Section 2 presents source-based mappings in detail. The idea of source-based mappings has been implemented as a framework called SCATR, which is described in Section 3. SCATR has been applied to a number of cases, which we report upon in Section 4. In Section 5 we compare source-based mappings with schema-based mappings, and propose a way to automatically check the safeness and strictness properties. Section 6 discusses related work.

2. Source-based Mappings

Figure 1 shows the general idea of source-based mappings. The left hand side is the domain of an analysis tool, while the

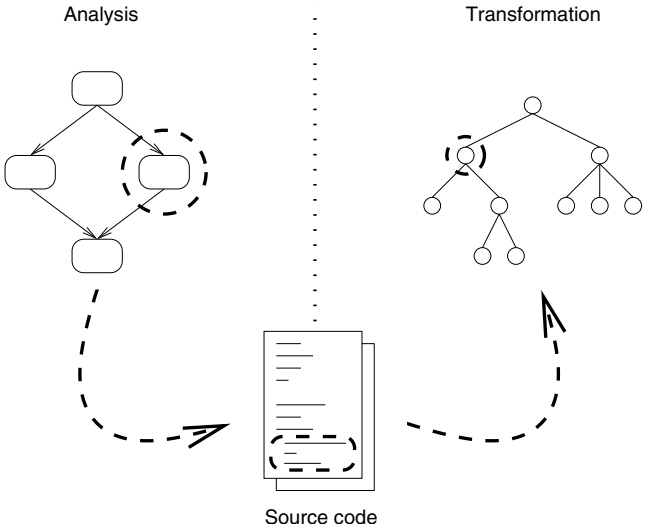

Figure 1. Source-based mappings.

```
for (i = 0; i < length; i++)                    1
{                                               2
  if (array[i] > max)                           3
  {                                             4
    max = array[i];                             5
  }                                             6
}                                               7
```

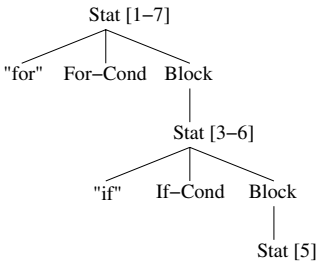

Figure 2. Source code example.

transformation domain resides on the right hand side. Both operate on the same body of source code. As is suggested by the figure, the tools work with different schemas.

The dashed circles and arrows show how a source-based mapping operates. First, an element of the schema used by the analysis tool is selected. We will refer to such an element as an *instance* of the schema used by the tool. An instance can be of a certain *type*, for example a PDG or AST node.

Subsequently, the selected instance is mapped to an appropriate area of the source code, along with the facts of interest associated with the instance. Next, the process is reversed in the transformation domain. The source code area obtained in the previous step is used to map the facts of interest to an appropriate instance of the schema used by the transformation tool.

If analysis and transformation operate on the same schema, and their mapping to and from the source code is identical, it is clear that a source-based mapping will allow facts about arbitrary instances to be exchanged. In practice, this situation is a rare exception, unless analysis and transformation are performed by the same tool. We are interested in the case where analysis and transformation are done by different tools, and possibly using different schemas, and therefore with a different source code correspondence.

A source-based mapping can be split into two functions, *down* and *up*. *Down* represents the arrow on the left hand side of Figure 1, while *up* represents the right hand side arrow. Given that we have fixed types S and T of instances in the analysis and transformation tools respectively, *down* and *up* have the following signatures:

$$\begin{aligned} down(S) &\longmapsto Area, \\ up(Area) &\longmapsto T, \end{aligned}$$

where *Area* refers to a source code area, e.g., a start line and column, paired with an end line and column. A source-based mapping then consists of the composition $up(down(s))$, where s is an instance of type S, and the result is an instance of type T.

Note that with the current definition, $down(s)$ yields a single area corresponding to s. However, elements of some representations may not be mappable to a single area of source code. For example, a usage dependency (e.g., an edge in a call graph) between modules may map to any of the call sites or variable accesses giving rise to the dependency. In these cases it may be desirable to allow $down(s)$ to yield a list of areas, and apply up to each area separately. We plan to investigate this matter in the future.

Consider the example source code in Figure 2, and its abridged AST representation. The Stat nodes in the AST are annotated with the line numbers that they correspond to. Suppose analysis results defined for PDG nodes are to be mapped to Stat nodes in the AST (i.e., S is fixed to PDG nodes, and T is fixed to Stat nodes). Let s be the PDG node representing the condition of the for loop, and let $down(s)$ yield the area a which spans line 1. Now $up(a)$ should yield the Stat node corresponding to the for loop, since a is included (only) in the area ([1–7]) of that node.

There may be some situations in which a source-based mapping is more problematic. First, the area yielded by $down(s)$ might correspond to more than one instance of type T. A common cause of this problem is recursion in grammars. For instance, expressions are typically defined recursively, and as a result, more than one expression may be defined at a source code area. In Figure 2, the statement `max = array[i];` at line 5 is nested within both the for and if statements. Consider that s is a PDG node representing the statement at line 5, and $down(s)$ is the area spanning line 5.

Now there are 3 Stat nodes t to which $up(down(s))$ could map, because line 5 is included within the area of any of the for, if, and assignment statements.

A partial solution to this problem would be to have more fine-grained source code correspondence within both analysis and transformation tools. For example, if the source correspondence of the AST node for the if statement in Figure 2 would consist of the lines 3, 4, and 6, instead of the entire range 3–6, then the if statement does not need to be considered as a target of the up of line 5.

Note that the $down$ function also needs to accommodate the finer-grained source correspondence. Since the up of line 5 no longer yields the AST node of the if statement, $down(s)$ has to output any of lines 3, 4 or 6 if s is a PDG node representing the if statement. Clearly, whether a finer-grained source correspondence can be used on one end is dependent on the other end. $Down$ and up have to be implemented in a compatible way, and therefore the implementor has to be aware of the source correspondences of both S and T.

Tools with an inaccurate source correspondence are therefore particularly problematic. However, some means are needed to cope with these inaccuracies in practice, since source correspondence within existing tools cannot always be easily improved. If the source correspondence at either end is not accurate enough to obtain a unique target for $up(down(s))$, a strategy has to be defined which implements a choice. For example, up could select the instance that is most specific to the area generated by $down(s)$. In Figure 2, up would then map line 5 to the assignment statement without a problem. Our SCATR framework (see Section 3) implements this strategy.

Another problem that may occur due to an inaccurate source correspondence is that the source code area $down(s)$ may not be associable with any instance of type T, because no such instance is defined at $down(s)$. This problem may also be caused by a bad choice of instance types, e.g., trying to map PDG nodes representing assignments to AST nodes representing if statements. The implementor of a source-based mapping has to make sure the instance types are chosen such that this problem cannot occur.

If the second problem (i.e., no instances of type T at $down(s)$) is not present, or in other words, if $up(down(s))$ is defined for all s from the domain, we call a source-based mapping *safe*. Furthermore, a source-based mapping that yields exactly one t for each s is called a *strictly safe* mapping. Both properties can be checked to hold given the instance types S and T, implementations of $down$ and up, and a body of source code. We discuss this further in Section 5.

3. SCATR

SCATR (short for Scaffolding And TRansformation, and pronounced as 'scatter') is a framework supporting the use of source-based mappings in the context of linking analysis and transformation tools. Scaffolding is a technique proposed by Sellink and Verhoef in [21], which constitutes the foundation of SCATR.

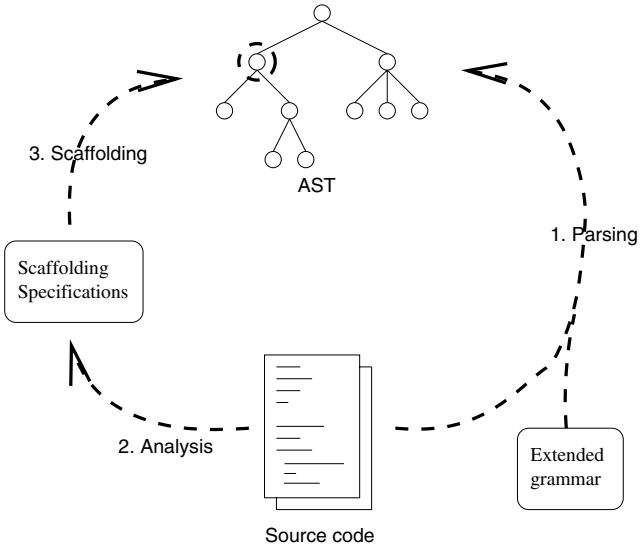

Figure 3. SCATR overview.

SCATR is not completely generic, in the sense that the target transformation tool is fixed; it is aimed at transformations expressed in ASF+SDF [4] only. Nevertheless, SCATR is not tied to a particular analysis tool. Furthermore, SCATR is independent of the language used in the source code, provided an SDF grammar for that language is available.

In terms of Figure 1, SCATR operates within the "Transformation" domain. Its purpose consists of inserting analysis results expressed as scaffolding specifications into the AST used by the transformation tool (i.e., ASF+SDF). Figure 3 gives an overview of SCATR. Three steps are performed to decorate an AST with analysis results.

1. Parsing with a grammar extended with support for *scaffolds*, resulting in an AST corresponding to the source code. Scaffolds are akin to parse tree annotations [20], and allow analysis results to be attached to nodes in the AST. The precise definition of scaffolds is discussed below.

2. Analysis results are generated by an appropriate analysis tool. The results of the tool are expressed as *scaffolding specifications*, which steer the process of inserting scaffolds in the AST. Scaffolding specifications are also discussed below.

3. Scaffolding is the final step in which the analysis results are inserted into the AST based on the scaffolding specifications.

In the remainder of this section we will first discuss the implementation of SCATR, followed by a discussion of design decisions underlying SCATR's architecture.

3.1. Implementation

Figure 4 presents the core modules of the ASF+SDF implementation of SCATR. The format used is SDF, which is similar to EBNF, except that the right and left hand sides of the grammar rules are swapped. Any parameters of a module are listed between square brackets next to the module's name. A parameter of an SDF module allows the user to specify a grammar non-terminal for which the module should be instantiated. The result of supplying an argument to a parameter is essentially a textual replacement of the occurrences of the parameter by its argument.

Extended-Language. The module ExtendedLanguage allows a grammar to be extended to facilitate scaffolding. It defines grammar productions that allow (one or more) Extensions before or after the Element of interest. The user can specify two parameters when using this module. Element is the grammar sort the user intends to extend. For instance, Statement would be specified if the user wishes to extend statements. Extension would normally be specified as Scaffold, but additional uses (e.g., comments, annotations) justify an additional layer of abstraction, as proposed by Sellink and Verhoef [21].

Scaffolder. The main module of SCATR defines the scaffolder function. The scaffolder function traverses its Program argument and inserts scaffolds to nodes of type Element according to a list of ScaffoldingSpecs supplied as the second argument.

The user of this module has to make sure the Program and Element parameters are set correctly. Program is to be instantiated as the top-level sort of the source code grammar, while the Element parameter should be set to the sort the user wishes to add scaffolds to.

Scaffolding-Spec. A ScaffoldingSpec specifies the insertion of a scaffold at a certain node in the AST. To select the target node the user specifies a Position, that is, line and column number, in the source file from which the AST was derived. The scaffolder will attach the scaffold to the lowest node in the AST that includes the position in its source range. Whether the scaffold is added to the left or to the right of the selected node is determined by the Type, i.e., respectively `before` or `after`.

Scaffold. The syntactical definition of a scaffold resides in this module. This definition is loosely based on Sellink and Verhoef's definition in [21]. Scaffolds can contain nested lists of named data, which can be of various sorts. By default, Strings are allowed as Scaffold-Data, but the user can add custom sorts by instantiating the Ext-Scaffold-Data parameter.

3.2. Architecture

The source-based mapping for which SCATR was designed consists of source code positions, that is, pairs of line and column numbers. The analysis tool is expected to map an instance of its schema (e.g., a PDG node in CodeSurfer) to a single source code position (*down* function in Section 2). SCATR will attempt to map this source code position to an appropriate node in an AST maintained by ASF+SDF (up function in Section 2). The analysis tool is burdened with making sure that the source-based mapping obtained is safe, i.e., it must ensure that an AST node of the selected sort is defined at the source code positions it exports.

Scaffolding. Determining which nodes are extended with a scaffold depends on two sources of information. First, the user of the SCATR framework specifies the type[1] of AST nodes that can receive a scaffold. For instance, the user may choose to add scaffolds to Statement nodes, if Statement is a sort defined by the language grammar. In Subsection 3.1 we discuss how the user achieves the sort selection.

Second, the scaffolding specification lists source code positions paired with scaffolds containing data. The scaffolder function adds a scaffold to a node if and only if the node is of the selected sort, and the source code position specified with the scaffold is included in the source code area spanned by the node.

Note that this process requires that the target AST is fully decorated with source position information, that is, each node in the AST can be mapped to its corresponding area within the source code.

One intricacy of the scaffolding process remains to be explained. A source code position can point to more than one node of the user selected sort. In C, for example, nested statements, or expressions, can cause this effect. SCATR ensures that a *strict* (see Section 2) mapping is obtained through two design decisions. The AST is traversed in a bottom-up fashion, and a scaffold is inserted at most once. In effect, a scaffold is added to the most specific (or lowest) AST node of the user selected sort, that includes the specified source code position in its area. This behavior implemented by SCATR may not always be desirable (though it has been for our purposes).

The number of AST nodes that are pointed to by the source-based mapping could possibly be reduced by improving upon the accuracy of source code positions. Source code areas could alternatively be used to create a source based mapping. A source code area more accurately describes the source representation of an instance by specifying the line and column numbers of the start and end of the instance. We discussed this solution in Section 2, and have shown that the use of a finer grained source correspondence on one end

[1]The type of an AST node corresponds to a grammar sort or non-terminal.

```
module Extended-Language [ Element Extension ]

Extension+ Element                              →  Element
Element Extension+                              →  Element
```

```
module Scaffolder [ Program Element ]

scaffolder ( Program, Scaffolding-Spec* )       →  Program
```

```
module Scaffolding-Spec

"begin" Scaffold Type Position "end"            →  Scaffolding-Spec
"before" | "after"                              →  Type
"(" Natural Natural ")"                         →  Position
```

```
module Scaffold [ Ext-Scaffold-Data ]

"SCAFFOLD" "[" Scaffold-Data* "]"               →  Scaffold
Data-Name "[" Scaffold-Data* "]"                →  Scaffold-Data
[A-Z_]+                                         →  Data-Name
String                                          →  Scaffold-Data
Ext-Scaffold-Data                               →  Scaffold-Data
```

Figure 4. SDF excerpts of the core modules of SCATR.

(here in the analysis tool) requires changes in the other end (here the transformation tool). For flexibility SCATR uses the relatively inaccurate source code positions, and deals with multiple matching nodes by picking the most specific node.

Grammar Extension. In order for scaffolds to be added to AST nodes, the grammar used to parse the source code needs to be extended such that nodes of interest (i.e., of type T in terms of Section 2) can be preceded or followed by nodes representing scaffolds. [2] SCATR provides for a flexible way of extending grammars. The module Extended-Language adds two grammar productions to extend the sort of interest such that it can be preceded and followed by scaffolds. The details of this module are presented in Subsection 3.1.

Lexical Scaffolding. Scaffolding as defined by Sellink and Verhoef [21] operates slightly differently. Their approach extends the target grammar much more extensively, by allowing scaffolds in front of each terminal (occurrence of a lexical sort). This has the advantage that scaffolds can also be added directly to the source code itself, followed by an invocation to the parser to obtain a scaffolded AST. In our simple approach to grammar extension this results in many ambiguities during parsing.

An advantage of our approach is that the scaffolding process inserts scaffolds at exactly the nodes of interest in the AST. This is beneficial for the purpose of specifying transformations based on the scaffolds, as no extra work has to be done to locate the scaffolds (if any) associated with the node. Sellink and Verhoef's approach causes the scaffolds to be added as leaves in the AST, possibly a long way from the nodes of interest. Without additional support for locating scaffolds in the AST, this is an unpractical situation for the specification of transformations. Kort *et al.* provide methods that are capable of locating scaffolds, and dealing with them in transformations [14].

Finally, one could argue that simple grammar extension could suffice if one would lexically insert *bracketed* scaffolds. A bracketed scaffold surrounds the source region it applies to with brackets, so that no ambiguity arises during parsing. A similar approach is taken by source code factors [16]. As it turns out, lexically inserting bracketed scaffolds is not practical. The analysis tool exporting its results would then need to generate the positions of the brackets in a way that is lexically compatible with the grammar used by the transformation tool. In our approach, the analysis tool can suffice by generating a position that it knows to lie within the source area of an AST node of the desired sort (safeness).

[2] In systems which do not require AST transformations to be syntax preserving such grammar modification may be unnecessary.

4. Applications

The SCATR framework is currently being used in several real transformation tasks. These tasks consider components of a 10 million line C system, developed and maintained by ASML, a Dutch manufacturer of lithography solutions. The tasks are related to our earlier work on (crosscutting) concern isolation [5], and consist of elimination of concern code, and insertion of domain annotations (among others). These transformations are required in a larger migration effort toward aspect-oriented technology.

Source-to-source transformations are desired in these cases, since developers have to be able to work with the transformed code. Specifically this requires the abilities to parse code in the presence of C preprocessor directives (including macros), and to preserve comments and white-space. Due to the availability of an SDF grammar for ANSI C extended with preprocessing directives and rewriting with layout capability [24], the ASF+SDF Meta environment [4] is used to implement the transformation tasks. The C grammar was modeled strictly after the ANSI C specification, and extended with support for the specific preprocessor use within ASML.

Several analyses required to identify concern code have previously been implemented [5] as plugins to GrammaTech's CodeSurfer [2]. Since these analyses are not trivial, and significant effort would be needed to re-implement them in ASF+SDF, the choice was made to reuse the CodeSurfer plugins. SCATR was developed to solve the problem of leveraging CodeSurfer's analysis results in transformations expressed in ASF+SDF.

SCATR has been used for the transformation of two components, CC1 and CC2, consisting of 32,402 and 17,716 non-blank lines of code, respectively. Efforts are currently ongoing to apply SCATR to 10 components, totalling approximately 2 million lines of code.

4.1. Concern Code Elimination

The first transformation task we consider consists of the elimination of code belonging to a number of concerns:

- **Tracing**. Dynamic execution tracing of each function such that the values of input and output parameters can be inspected.
- **Timing**. Collection of timings for each function execution.
- **Function Naming**. Each function has a local variable which holds a string representing the function's name. These strings are used within tracing and logging calls.
- **Parameter Checking**. Pointer parameters of functions should not be NULL before they are referenced, each function therefore has to implement checks. The parameter checking concern is discussed in detail in [5].

```
THXAtrace(CC,                                          868
  THXA_TRACE_INT,                                      869
  func_name,                                           870
  ">␣(read_fd=%d,␣timeout=%d)",                        871
  read_fd,                                             872
  timeout);                                            873
```

Figure 5. Example tracing call.

```
begin
 SCAFFOLD["TRACING"]
 before
 (868 0)
end
```

Figure 6. Scaffolding specification for a single tracing call.

The instantiation of SCATR for the elimination of these concerns is very similar in all cases, therefore we suffice with a discussion of the elimination of the tracing concern in this paper. The tracing concern consists of calls to a tracing function, where the arguments are the values of either input or output parameters. An example is shown in Figure 5.

A CodeSurfer plugin was previously developed to identify all the tracing calls for all functions. The result consists of a set of PDG nodes representing the calls to the tracing function. Furthermore, a utility script was developed to export these PDG nodes along with the fact that they belong to the tracing concern, into SCATR's scaffolding specification format (see Section 3). An example scaffolding specification is shown in Figure 6. It states that a scaffold of the form `SCAFFOLD["TRACING"]` should be added before the instance at source code line 868, column 0. This source code position corresponds to the first character of the tracing call.

Effectively this export script implements the *down* function that was described in Section 2. The other part of the source-based mapping, the *up* function, is implemented by SCATR. Function calls are parsed as Statements in our SDF C grammar, thus instantiating SCATR for this task starts by extending the C grammar such that scaffolds can be added to Statement nodes. As was explained in Section 3, this is done through the parameters of the module ExtendedLanguage (see Figure 4).

The next step consists of the invocation of the scaffolder function, with the (parsed) source code and all generated scaffolding specifications as arguments. The result is an AST in which all the Statement nodes pointed to by the scaffold-

```
SCAFFOLD["TRACING"]<<THXAtrace(CC,
          THXA_TRACE_INT,
          func_name,
          ">␣(read_fd=%d,␣timeout=%d)",
          read_fd,
          timeout);>>
```

Figure 7. Tracing call decorated with a scaffold.

```
int CCCN_Wait(int read_fd,                          549
              int timeout)                          550
__trace__(in (read_fd timeout) out ())              551
{                                                   552
  . . .                                             553
}                                                   554
```

Figure 8. Tracing annotation.

ing specifications are decorated with a scaffold. Figure 7 shows how this would look if a decorated node was pretty printed.

Finally, the AST is traversed one more time by a function that removes all nodes decorated with a specific scaffold. In this case the traversal would look for tracing scaffolds, but for the other concerns the scaffolds contain the respective names of the concerns.

The source-based mapping we defined in this case works because it is strict and safe, as defined in Section 2. First, safeness holds because in the ANSI C grammar the first character of the name of the called function is guaranteed to point to an AST node of type Statement. Second, the mapping is also strict, due to SCATR's strategy of selecting the most specific node of type Statement. The Statement node representing the tracing call will always be lower in the AST than Statement nodes representing any surrounding statements. For now these properties follow from the structure of the grammar, and SCATR's selection strategy. In the future we would like to implement a tool to check whether these properties hold given a body of source code, and a defined source-based mapping.

4.2. Insertion of Domain Annotations

After all tracing code has been eliminated (see previous Subsection), a compile-time weaver is responsible for regenerating the tracing functionality. As it turns out, the tracing concern requires some non-trivial analysis to figure out which function parameters are used as input and which are used as output parameters. Since it would be costly to integrate this analysis into the build process, it was decided to perform a one-time analysis of the source code, and insert the results (i.e., input / output characteristics) into the code as *domain annotations*. In our case, the domain annotations are specifically targeted at a compile-time weaver for C, WeaveC [1]. Here we will discuss the use of SCATR in the annotation process.

Again, the analysis is performed by a CodeSurfer plugin, and results in a list of input and output parameters for each function (PDG). Figure 8 gives an example of how a domain annotation should be inserted in the source code at line 551. The annotation has been inserted after the function signature, and before the defining block of the function. It shows

```
begin
 SCAFFOLD [IN [read_fd timeout] OUT []]
 after
 (550 25)
end
```

Figure 9. Scaffolding specification for temporary scaffolds.

that this function (`CCCN_Wait`) has two input parameters, `read_fd` and `timeout`, and no output parameters.

The insertion process works by first inserting temporary scaffolds containing the analysis results, and then translating the scaffolds into the desired domain annotation format. SCATR again requires the selection of the grammar sort to which scaffolds need to be added. Since the annotation has to be inserted after the function signature, the appropriate sort is Declarator, which spans the area from the start of the function name (`CCCN_Wait` in Figure 8), up to and including the closing parenthesis of the signature. A CodeSurfer script is used to implement *down* by generating a scaffolding specification with the input/output information wrapped in a scaffold, and the source code position corresponding to the closing parenthesis of the function signature. An example is shown in Figure 9.

Similar to the result in Figure 7, running the scaffolder function on the source code with the annotation scaffolding specifications results in function signatures with scaffolds appended to them. The final step then consists of a traversal which trivially translates the present scaffolds into the tracing annotations shown in Figure 8.

Safeness and strictness of the mapping used in this case follows by the same argument we used before. According to the ANSI C grammar, the closing parenthesis of a function signature matches exactly one node of sort Declarator.

5. Discussion

Source-based mappings vs. schema-based mappings

The defining feature of source-based mappings is that they locate the instances of interest through pointers in the source

code itself. Other approaches exist to solve the location problem, that do not utilize source correspondences at all (or as much). These *schema-based* approaches locate instances of interest in the target (transformation) tool through queries expressed using the target schema. HSML is, in essence, such an approach [7], since it allows maintenance hot spots to be identified through complex queries expressed using the grammar of the target language. Other examples are the various query languages for XML documents, e.g. XPath or XQuery.

Let's consider how a schema-based mapping could be used in the context of linking an analysis and a transformation tool. The setting is the same as defined in Section 2, i.e., we want to map analysis results for instances of type S to instances of type T in a transformation tool. The analysis and transformation tools have different schemas of the source code. What alternatives to the $down$ and up functions would need to be implemented?

To start with $down$, recall that its purpose is to map an instance s to an appropriate area of source code, such that up can map that area to an instance of type T. The analogue of $down$ in a schema-based mapping then consists of a function that maps s to an appropriate expression in the schema used by the transformation tool. Subsequently, the analogue of up is tasked with interpreting this expression and applying the results to the matching instances. For instance, an expression pointing out the assignment statement (at line 5) in Figure 1 could consist of *Stat – Block – Stat [0] – Block – Stat [0]*, where *Stat [i]* would refer to the i-th statement within a block.

The question that arises is what knowledge is required for the implementation of a schema-based mapping, and how does this compare to a source-based mapping? Creating an expression that points out an instance of interest requires knowledge of the target schema. For example, the expression above can only be created if it is known that the Stat node of interest is the 0-th child of its parent Block node containing it, which is a child of a Stat node itself, and so forth, all the way up to the top Stat node. As a result, creating such an expression requires knowledge of the target schema from the top down to at least the type of the instances of interest. Possibly this knowledge requirement can be mitigated by designing a query language that allows abstraction, but we conjecture that at least all the containment relations must still be known in order to accurately point out the instances of interest.

A source-based mapping requires different knowledge of the target schema. $Down(s)$ has to yield a source code area that corresponds to an instance of type T at the transformation end (safeness). Therefore, implementing $down$ cannot be done without knowledge of the source correspondence of instance type T. Recall the example explained in Section 2, where a finer-grained source correspondence within the transformation tool required $down$ to be changed accordingly. However, as long as the safeness property can be guaranteed, $down$ does not have to re-generate the exact source correspondence of the transformation tool.

Additional awareness is needed for guaranteeing the strictness property, i.e., making sure that at most one instance of type T is the target of $up(down(s))$. This problem can occur if $down(s)$ is included in the source correspondence of more than one instance of type T. Therefore, implementing a source-based mapping requires knowledge of overlapping source correspondences of instances of type T. If the problem cannot be evaded by changing $down(s)$ to generate a more specific source code area, a strategy will have to be implemented in up to make a choice, as was discussed in Section 2.

In summary, a schema-based mapping requires detailed knowledge of the target schema. It may be required to know the definition of other types of instances than the type of interest, because containment relations need to be traversed from the top down leading to the instance of interest. In contrast, the knowledge a source-based mapping needs of the target schema is limited to the type of interest only. However, it is required to be aware of its source corresponce, and possible overlapping source correspondences of instances of the type of interest.

Automatically checking safeness and strictness

The use of source-based mappings in real transformation tasks may benefit from some form of automated verification. In particular, checking the safeness and strictness properties could be a good starting point. Fortunately, these properties can be checked automatically for a fixed body of source code by the following process. First, the domain of the source-based mapping is established. All instances s of type S belong to the domain. For each s then $up(down(s))$ is performed, yielding a list of instances of type T. If this list is empty, no instance of type T was found defined at $down(s)$, the source-based mapping is not safe, and an error must be reported. If the list contains exactly one instance t, the mapping is safe and strict for s. Finally, a list of length 2 and more indicates that multiple instances are defined at $down(s)$, and the mapping is not strict, resulting in an error. After all s have been checked, the complete mapping is only safe and strict if no errors have been reported.

For some schemas, the safeness and strictness properties could even be determined for all possible bodies of source code. For now this remains the area of future research.

6. Related Work

Tool interoperability

The topic of tool interoperability has been widely discussed in the community [8, 9]. A large number of proposals exist in the literature that contribute a solution to interoperability issues. Among others, technologies like the ToolBus [3], OASIS [13], and IDL [22] provide architectures for integration of tools. Communication is an essential part of tool interoperability, and as such a number of data interchange formats have been defined. Examples are GXL [12], a graph-based format, ATerms [23], and RSF [18]. Technologies like these provide tool interoperability solutions at a different level than source-based mappings. In terms of [8], these technologies provide protocols, (data) marshalling, or representations. Source-based mappings are aimed at solving the identification (of source elements) problem.

Markup and annotations

Source code markup is a technique that has been used in many different contexts. Here we focus only on those approaches that are closely related to source-based mappings. Scaffolding by Sellink and Verhoef [21] is proposed to be used to store intermediate results of transformations, and share results between tools via markup in the source code. However, they do not explicitly focus on the issue of tools using different schemas. Source code factors by Malton *et al.* [16] is an approach that is very similar to scaffolding, as it also marks up the source code with intermediate analysis and transformation results.

HSML by Cordy *et al.* [7] is a markup approach that allows maintenance hot spots to be defined as queries expressed in the target schema. In that sense it is a schema-based mapping as defined in Section 5, except that the results of the mapping are also made visible in the source code through markup. The difference with source-based mappings consists of the extensive use of the target schema by HSML. A source-based mapping is less dependent on the target schema, but more dependent on the source correspondence.

XML is a popular means to marking up source code. In [6], Cordy proposes a method to markup source code with task-specific XML. By employing agile parsing (possibly combined with island grammars [17]), the source code grammar can be adjusted to focus the source markup to those pieces of source code that are interesting to the task at hand. In our SCATR framework, pretty printing the scaffolded AST has the same result, since scaffolds are only added to those nodes that are interesting within a transformation task. [19] instead proposes to markup all the source code with XML corresponding to its AST, resulting in a verbose representation.

Tool interoperability schemas

A number of schemas have been proposed specifically for the purpose of tool interoperability. The Dagstuhl Middle Metamodel (DMM) [15] is aimed at object-oriented and procedural languages, and can further be extended by the user of the schema. Columbus [10] is a specific schema for C++ programs. Both Columbus and DMM are expressed as UML diagrams. Holt *et al.* [11] instead use an E/R diagram to define a schema for Datrix, a software exchange format for C, C++ and Java programs.

7. Conclusion

In this paper we discussed source-based mappings, a technique to link analysis and transformation tools. The setting used for this discussion consisted of analysis and transformation tools that do not share a schema of the source code, and therefore reuse of analysis results by the transformation tool is not trivial. We defined two properties, safeness and strictness, that constitute a base of confidence in the mapping between two tools. These properties can be checked automatically for a given body of source code, allowing for a practical way of verification of a source-based mapping.

The idea of source-based mappings was implemented in the SCATR prototype tool, which allows analysis results to be mapped into ASTs produced by ASF+SDF. Two applications showed how this technology could be used in practice to implement transformation tasks such as concern code elimination or insertion of domain annotations.

An interesting link may exist between this work and island grammars [17], or agile parsing [6]. These technologies allow the easy adaptation of grammars to specific tasks. For instance, the grammar could be limited to defining only the statements that need to be removed, or the program elements that analysis results must be attached to.

Acknowledgements Thanks to Jurgen Vinju, Rob Economopoulos, Tijs van der Storm and Tom Tourwé for correcting drafts of this paper. This work has been carried out as part of the Ideals project under the auspices of the Embedded Systems Institute. This project is partially supported by the Dutch Ministry of Economic Affairs under the Senter program.

References

[1] WeaveC. `http://sourceforge.net/projects/weavec/`.

[2] Paul Anderson, Thomas W. Reps, Tim Teitelbaum, and Mark Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, 2003.

[3] Jan A. Bergstra and Paul Klint. The discrete time TOOLBUS — a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, 1998.

[4] M. van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2), 2001.

[5] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, September 2005.

[6] James R. Cordy. Generalized selective XML markup of source code using agile parsing. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC)*, pages 144–153. IEEE Computer Society, May 2003.

[7] James R. Cordy, Kevin A. Schneider, Thomas R. Dean, and Andrew J. Malton. HSML: Design directed source code hot spots. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*, pages 145–156. IEEE Computer Society, May 2001.

[8] James R. Cordy and Jurgen J. Vinju. How to make a bridge between transformation and analysis technologies? In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Dagstuhl Seminar Proceedings*, number 05161, Schloss Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum (IBFI).

[9] J. Ebert, K. Kontogiannis, and J. Mylopoulos, editors. *Dagstuhl Seminar Interoperability of Reengineering Tools*, Schloss Dagstuhl, Germany, January 2001. Internationales Begegnungs- und Forschungszentrum (IBFI).

[10] Rudolf Ferenc and Árpád Beszédes. Data exchange with the columbus schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 59–66. IEEE Computer Society, March 2002.

[11] Richard C. Holt, Ahmed E. Hassan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/R schema for the datrix C/C++/Java exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE)*, pages 284–286. IEEE Computer Society, November 2000.

[12] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171, November 2000.

[13] Dean Jin and James R. Cordy. Ontology-based software analysis and reengineering tool integration: The OASIS service-sharing methodology. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 613–616. IEEE Computer Society, September 2005.

[14] Jan Kort and Ralf Lämmel. Parse-tree annotations meet re-engineering concerns. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 161–170. IEEE Computer Society, 2003.

[15] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, 2004.

[16] Andrew J. Malton, Kevin A. Schneider, James R. Cordy, Thomas R. Dean, Darren Cousineau, and Jason Reynolds. Processing software source text in automated design recovery and transformation. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*, pages 127–134. IEEE Computer Society, May 2001.

[17] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*, pages 13–22. IEEE Computer Society, October 2001.

[18] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[19] James F. Power and Brian A. Malloy. Program annotation in XML: A parse-tree based approach. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 190–198. IEEE Computer Society, October 2002.

[20] James M. Purtilo and John R. Callahan. Parse tree annotations. *Communications of the ACM*, 32(12):1467–1477, 1989.

[21] M. P. A. Sellink and C. Verhoef. Scaffolding for software renovation. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 161–172. IEEE Computer Society, February 2000.

[22] Richard T. Snodgrass and Karen Shannon. Supporting flexible and efficient tool integration. In *Advanced Programming Environments, Proceedings of an International Workshop*, Lecture Notes in Computer Science, pages 290–313. Springer, June 1986.

[23] Mark van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software — Practice and Experience*, 30(3):259–291, 2000.

[24] Jurgen J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, University of Amsterdam, 2005.