Fine Slicing

Theory and Applications for Computation Extraction

Aharon Abadi*, Ran Ettinger, and Yishai A. Feldman

IBM Research – Haifa
{aharona,rane,yishai}@il.ibm.com

Abstract. Software evolution often requires the untangling of code. Particularly challenging and error-prone is the task of separating computations that are intertwined in a loop. The lack of automatic tools for such transformations complicates maintenance and hinders reuse. We present a theory and implementation of fine slicing, a method for computing executable program slices that can be finely tuned, and can be used to extract non-contiguous pieces of code and untangle loops. Unlike previous solutions, it supports temporal abstraction of series of values computed in a loop in the form of newly-created sequences. Fine slicing has proved useful in capturing meaningful subprograms and has enabled the creation of an advanced computation-extraction algorithm and its implementation in a prototype refactoring tool for Cobol and Java.

1 Introduction

Automated refactoring support is becoming common in many development environments. It improves programmer productivity by increasing both development speed as well as reliability. This is true in spite of various limitations and errors due to insufficiently detailed analysis. In a case study we performed [1], we recast a manual transformation scenario[1] as a series of 36 refactoring steps. We found that only 13 steps out of these 36 could be performed automatically by modern IDEs. Many of the unsupported cases were versions of the Extract Method refactoring, mostly involving non-contiguous code.

The example of Figure 1(a) shows the most difficult case we encountered. At this point in the scenario, we want to untangle the code that outputs the selected pictures to the HTML view (lines 1, 7, and 9) from the code that decides which pictures to present. The subprogram that consists only of these three lines does not even compile, because the variable `picture` is undefined. However, the more serious defect is that it does not preserve the meaning of the original program, since the loop is missing, and this program fragment seems to use only one picture. To preserve the semantics, the extracted subprogram needs to receive the picures to be shown in some collection, as shown in Figure 1(b). The rest of the code needs to create the collection of pictures and pass it to the new

* Also at Blavatnik School of Computer Science, Tel Aviv University.

[1] `http://www.purpletech.com/articles/mvc/refactoring-to-mvc.html`

J. de Lara and A. Zisman (Eds.): FASE 2012, LNCS 7212, pp. 471–485, 2012.

(a)

```
1 out.println("<table>");
2 int start = page * 20;
3 int end = start + 20;
4 end = Math.min(end, album.getPictures().size());
5 for (int i = start; i < end; i++) {
6     Picture picture = album.getPicture(i);
7     printPicture(out, picture);
8 }
9 out.println("</table>");
```

(b)

```
1 public void display(PrintStream out, int start,
2                     int end, Queue<Picture> pictures) {
3     out.println("<table>");
4     for (int i = start; i < end; i++)
5         printPicture(out, pictures.remove());
6     out.println("</table>");
7 }
```

(c)

```
1 int start = page * 20;
2 int end = start + 20;
3 end = Math.min(end, album.getPictures().size());
4 Queue<Picture> pictures = new LinkedList<Picture>();
5 for (int i = start; i < end; i++) {
6     Picture picture = album.getPicture(i);
7     pictures.add(picture);
8 }
9 display(out, start, end, pictures);
```

Fig. 1. (a) A program that tangles the logic of fetching pictures to be shown with their presentation. (b) Presentation extracted into a separate method. (c) Remaining code calls the new method.

method, as in Figure 1(c). This transformation is crucial in the scenario, as it forms the basis of the separation of layers. The code that deals with the HTML presentation is now encapsulated in the `display` method, and can easily be replaced by another type of presentation.

One possibility for specifying the subprogram to be extracted is just to select a part of the program, which need not necessarily be contiguous. In fact, the subprogram need not even contain complete statements; it is quite common to extract a piece of code replacing some expression by a parameter [1, Fig. 2]. However, in most cases the subprogram to be extracted is not some arbitrary piece of code, but has some inherent cohesiveness. In the example of Figure 1, the user wanted to extract the computations that write to the `out` stream, but without the computations of `start`, `end`, and `picture`.

This description is reminiscent of program slicing [17]. A (backward) slice of a variable in a program is a subprogram that computes that variable's value. The smallest such subprogram is of course desirable, although it is not computable

in general. However, a full slice is often too large. In our example, the slice of `out` at the end of the program in Figure 1(a) is the whole program, since all of it contributes to the output that will be written to `out`. In general, a slice needs to be closed under data and control dependences. Intuitively, one statement or expression is data-dependent on another if the latter computes a value used by the former. A statement is control-dependent on a test[2] if the test determines whether, or how many times, the statement will be executed.

In this paper we present the concept of *fine slicing*, a method that can produce executable and extractable slices while restricting their scopes and sizes in a way that enables fine control. This is done by allowing the user (or an application that uses fine slicing) to specify which data and control dependences to ignore when computing a slice. In particular, the subprogram we wanted to extract from Figure 1(a) can be specified as a fine slice of the variable `out` at the end of the program, ignoring data dependences on `start`, `end`, and `picture`.

Our fine-slicing algorithm will add to the slice control structures that are needed to retain its semantics, even when these control structures embody dependences that were specified to be ignored. For example, suppose that instead of line 1 in Figure 1(a) the following conditional appeared:

```
if (album.pictureSize() == SMALL)
    out.println("<table cellspacing='10'>");
else
    out.println("<table>");
```

The conditional will be added to the fine slice even if control dependences on it were specified to be ignored, since the subprogram that does not contain it will always execute both printing statements instead of exactly one of them. However, the data that the test depends on will not be included in the fine slice in that case. This part of the fine-slicing algorithm is called *semantic restoration*.

Fine slicing has many applications. For example, it can be used to make an arbitrary subprogram executable by adding the minimum necessary control structures. (This can be construed as a fine slice that starts from the given subprogram and ignores all dependences it has on other code.) In this paper we show in detail how fine slicing can be used in a generalization of Extract Method, which we call *Extract Computation*, that can handle non-contiguous code and other difficult transformations.

The contributions of this paper include:

- a theory of fine slicing, with an oracle-based semantics;
- an algorithm for fine slicing, including semantic restoration;
- a demonstration of the utility of fine slicing for the Extract Computation refactoring; and
- a prototype implementation of fine slicing and Extract Computation for Cobol and Java in Eclipse.

[2] In this context, we use the word "test" to refer to any conditional branch in the program's flow of control.

1.1 Fine Slicing

Slicing algorithms typically use some representation of the program with pre-computed data and control dependences. In order to compute a (backward) slice, the algorithm starts from an initial slice containing the user-selected locations (also called the *slicing criteria*). It then repeatedly adds to the slice any program location on which some part of the current slice has a data or control dependence. The final slice is available when the process converges. In the case of backward slices, the result is executable. (Forward slices are usually not executable.)

A fine slice can be computed in the same way, except that those dependences specified to be ignored are not followed. This, however, can result in a slice that has compilation errors, is not executable, or does not preserve the original semantics. This may be due to two types of problems: missing data, and missing control. Missing data manifests itself as the use of a variable one or more of whose sources (the assignments in the original program from which it receives its value) are unavailable. We consider the variable to have missing sources when it is disconnected from its sources in the original program, even if the subprogram appears to supply other sources for it.

Missing control creates control paths in the subprogram that are different from those in the original program, as in the case of the two table-header printing statements that would appear to be executed sequentially without the surrouding conditional.

We offer two different ways to deal with these problems. For missing data, we provide an oracle-based semantics, where the oracle supplies the missing values. In order to make the subprogram executable, the oracle can be simulated by appropriate variables or sequences. For missing control, our semantic restoration algorithm adds to the subprogram just those control structures that are necessary to make it preserve its original semantics. However, the data for these control structures is *not* added, being supplied by the oracle instead. These notions are formalized in Section 2.

While fine slicing can be used directly by a user using a tool that displays slices based on various criteria, we expect fine slicing to be used as a component by other applications, such as Extract Computation. In particular, we do not expect users to directly specify control dependences to be ignored. Data dependences are much easier for users to understand, and our graphical user interface for Extract Computation provides a convenient way to specify data dependences to be ignored.

1.2 Extract Computation

The example of Figure 1 shows the two types of difficulties involved in untangling computations. First, it is necessary to identify the relevant data sources as well as the control structures the subprogram to be extracted needs in order to preserve its semantics. This information can be used to generate the method encapsulating the extracted subprogram: the control structures are included in the new method, and the data sources are passed as parameters. This is achieved by the fine slicing algorithm.

Second, it is necessary to modify the original code; among other things, it needs to prepare any parameters and call the new method. As shown in the example, some parts of the original code (such as the loop) need to be duplicated. A *co-slice*, or complement slice [6], is the part of the program that should be left behind once a slice has been extracted from it. As shown above, the co-slice may contain some code that is also part of the extracted fine slice. It turns out that a co-slice is a special case of a fine slice, which starts from all locations not extracted and ignores data values to be returned. In the example, it is the slice from lines 2–4 and 6 of Figure 1(a), ignoring the final value of `out`.

The Extract Computation refactoring extracts the selected code and replaces it with an appropriate call. It computes the two fine slices and determines where the call to the extracted code should be placed. Some of the parameters need to aggregate values computed through a loop. The Extract Computation algorithm determines which data values are multiple-valued, and creates the code to generate the lists containing these values.

The Extract Computation refactoring is general enough to support all the cases in our case study [1] that were not supported by existing implementations of Extract Method. In particular, it can support the extraction of non-contiguous code in several varieties. In addition to the example above, demonstrating the extraction of part of a loop with the minimal required duplication of the loop, they include: extracting multiple fragments of code; extracting a partial fragment, where some expressions are not extracted but passed as parameters instead; and extracting code that has conditional exits (caused by `return` statements in the code to be extracted) [2]. A detailed description of the algorithm appears in Section 3.

2 A Theory of Fine Slicing

We assume a standard representation of programs, which consists of a control-flow graph (CFG), with (at least) the following relationships defined on it: domination and post-domination, data dependence and control dependence. We use $\mathrm{dflow}_P(d_1, d_2)$ to denote the fact that a variable definition d_1 reaches the use d_2 of the same variable in program P. We require that variable definitions include assignments to the variable as well as any operation that can modify the object it points to. In the example of Figure 1, any method applied to `out` can modify the output stream, and needs to be considered a definition of `out`. We assume a standard operational semantics, in which each state consists of a current location in the CFG of the program, and an environment that provides values of some of the program's variables. We also assume some mechanism that makes states in the same execution unique; for example, a counter of the number of times each node was visited.

We extend this representation to *open programs*, in which some variable uses can be marked as disconnected; these have no definitions reaching them. After defining the notion of a subprogram, we extend the operational semantics with oracles for disconnected variables, show how an oracle for an open subprogram is

induced from the corresponding execution of the original program, and formalize fine slices as open subprograms that, given the induced oracle, compute the same values for all variables of interest.

Definition 1 (subprogram). *A (possibly open) program Q is a* subprogram of *a program P if*

1. *all CFG nodes of Q belong to P;*
2. *for every variable definition point d and variable use point u in Q that is not disconnected,* $\mathrm{dflow}_Q(d,u)$ *is true iff* $\mathrm{dflow}_P(d,u)$ *is true; and*
3. *there is an edge from CFG node $n_1 \in Q$ to $n_2 \in Q$ iff there is a non-empty path from n_1 to n_2 in P that does not pass through other nodes in Q, except when n_1 is the exit node of Q and n_2 is the entry node of Q.*

We define a state s of P and a state s' of Q to be equivalent with respect to the connected variables (denoted $E(s,s')$) if their environments coincide on all common variables, except possibly for those that are disconnected in the current node. The initial states of Q will be restricted to those that are equivalent to states of P that can be reached by executions of P without visiting any nodes of Q before reaching those states.

An oracle $O(s,u)$ for an open program Q is a partial function that provides values for each disconnected variable u at each possible state s of the program. An execution of an open program is defined by extending the operational semantics of programs so that at any point where the value of a disconnected variable u is required in an execution state s, the value used will be $O(s,u)$. If this value is undefined, the execution is deemed to have failed.

The execution of P provides the oracle that can be used to supply the missing values in a corresponding execution of Q. Denote a single step in the operational semantics of P by $\mathrm{step}_P(s)$, and the value of a variable use u in state s by $\mathrm{env}_P(s,u)$.

Definition 2 (induced oracle). *The* oracle induced by a program P on an open subprogram Q of P from an initial state s_0 of P *is defined as follows: if* $\mathrm{step}_P^k(s_0) = s$ *for some $k \geq 0$, the current location in state s belongs to Q, u is a disconnected variable use in the current location, and $E(s,s')$, then* $\mathrm{oracle}_Q^{P,s_0}(s',u) = \mathrm{env}_P(s,u)$.

Under this definition, any open subprogram of P is executable with an oracle and preserves the behavior of P.

Theorem 1 (correctness of execution with oracle). *Let Q be an open subprogram of P, s_0 an initial state of P, and q_0 the corresponding initial state of Q (assuming one exists). If P halts*[3] *(i.e., reaches its exit node) when started at p_0, then Q will also halt when started at q_0 with the induced oracle,* $\mathrm{oracle}_Q^{P,s_0}$, *and will compute the same values for all common variable occurrences.*

[3] It is possible to relax this condition to specify that P reaches a state from which it cannot return to Q.

This theorem does not imply that an arbitrary sub-graph of the CFG of P is similarly executable and semantics-preserving, since the theorem only applies to subprograms (Def. 1), whose structure is constrained to preserve the data and control flow of P. The semantic restoration algorithm can be applied to any collection of CFG nodes, and will complete it into a subprogram by adding the required tests, even when control dependences on them were specified to be ignored. This is a crucial feature that makes fine slices executable and semantics-preserving. However, the data on which this control is based may still be disconnected. Therefore, adding these tests will not require the addition of potentially large parts of the program involved in the computation of the specific conditions used in these tests.

There are different ways to specify the dependences to be ignored by a particular fine slice, but ultimately these must be cast in terms of a set D of variable uses to be disconnected, and a set C of control dependences to be ignored (each represented as a pair (t, n) where a node n depend on a test t). As usual, the slice is started from a set of slicing criteria, which we represent as a set S of nodes in the CFG of P.

Definition 3 (fine slice). *Let P be a program, S a set of slicing criteria, D a set of variable uses to be disconnected in P, and C a set of control dependences from P to be ignored. A* fine slice of P with respect to S, D, and C *is an open subprogram Q of P that contains all nodes in S, and in which every disconnected variable use d satisfies at least one of the following conditions:*

1. *the variable use d was allowed to be disconnected: $d \in D$; or*
2. *d is variable use in a test node t on which all control dependences from elements in the slice are to be ignored: if $n \in Q$ is control-dependent on t then $(t, n) \in C$.*

We now present an algorithm that computes fine slices. The algorithm accepts as inputs a program P, a set S of slicing criteria, a set D of input variable uses that are allowed to remain disconnected in the fine slice, and a set C of control dependences that may be ignored.

The algorithm consists of the following main steps:

1. Compute the core slice Q by following data and control dependence relations backwards in P, starting from S. Traversal of data dependences does not continue from variable uses in D, and traversal of control dependences does not follow dependences that belong to C.
2. (Semantic Restoration) Add necessary tests to make the slice executable.
3. Connect each node $n_1 \in Q$ to a node $n_2 \in Q$ iff there is a path from n_1 to n_2 in P that does not pass through any other node in Q.

As explained above, in order to turn the fine slice into a subprogram, it is necessary to add some tests from the original program even though all their control dependences have been removed. This is the purpose of the semantic-restoration step, which comprises the following sub-steps:

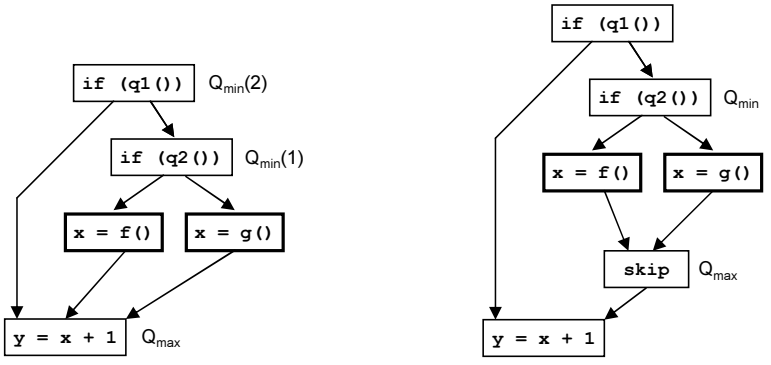

Fig. 2. Semantic restoration: original program (left); after separation of merges (right)

2.1 Let $Q_{\min}$ be the lowest node in P that dominates all the nodes of Q. Add $Q_{\min}$ to Q if it is not already there.
2.2 Let $Q_{\max}$ be the highest node in P that postdominates all the nodes of Q. Add $Q_{\max}$ to Q if it is not already there.
2.3 Add to Q all tests t from P that are on a path from some $q \in Q$ to $q' \in Q$ where q' is control-dependent on t, except when $q = Q_{\max}$ and $q' = Q_{\min}$. Do *not* add any data these tests depend on.
2.4 Repeat steps 2.1–2.3, taking into account the new nodes added each time, until there is no change.

In step 2.1, the "lowest" node is the one that is dominated by all other nodes in P that dominate all the nodes in Q. Similarly, in step 2.2 the "highest" node is the one that is postdominated by all other nodes in P that postdominate all nodes in Q. The smallest control context surrounding the fine slice is given by the part of the program between $Q_{\min}$ and $Q_{\max}$, and this determines the extraction context. This computation may need to be iterated for unstructured programs (including unstructured constructs in so-called structured languages); this is the purpose of step 2.4. When choosing $Q_{\max}$, the algorithm may need to add dummy nodes so as to separate merge points that are reachable from Q from those that are not. This will ensure that the new value of $Q_{\max}$ will not move $Q_{\min}$ to include unnecessary parts of the program. In the example of Figure 2, Q consists of the two assignments to `x` (with emphasized borders). On the left side of the figure, no separation of merges has been done. Initially, $Q_{\min}$ will be the second conditional (marked $Q_{\min}(1)$), and $Q_{\max}$ will be the assignment to `y`. This will force $Q_{\min}$ to move to the first conditional (marked $Q_{\min}(2)$), making Q contain the whole program. With the optimization of separating merges, shown on the right of the figure, $Q_{\max}$ is the new dummy node, and $Q_{\min}$ remains at the second conditional, making the fine slice smaller.

Step 2.3 excludes paths from $Q_{\max}$ to $Q_{\min}$, since these correspond to loops that contain the whole fine slice, and should not be included in it. When the fine slice is extracted, the call will appear inside any such loops.

Step 2.4 is only necessary for unstructured programs; with fully structured code a single iteration is always sufficient.

Theorem 2. *The fine slicing algorithm is well defined (i.e., nodes in steps 2.1 and 2.2 always exist), and produces a valid fine slice.*

Because the result of the algorithm is an open subprogram of P, it will compute the same values as P given the induced oracle (Theorem 1).

Theorem 3. *The worst-case time complexity of the fine-slicing algorithm is linear in the size of the program-dependence graph (i.e., the size of the control and data dependence relations).*

3 Extract Computation

The Extract Computation refactoring starts with a (possibly open) subprogram Q to be extracted from a program P. The subprogram can be a fine slice, chosen by the user or by another application; it can also be the result of applying semantic restoration to an arbitrary collection of statements. As a subprogram, it preserves the original semantics given the appropriate values from the induced oracle. Not all the code of Q can be removed from its original location, since some of it may be used for other purposes in P, as in the case of the common loop in Figure 1. The algorithm needs to compute the co-slice, replace the extracted code with an appropriate call, and implement the data-flow to the extracted code in the form of parameters and return values. Some of these values may be sequences, and the algorithm determines which they are and how to compute them.

The co-slice will contain all parts of the program that have not been extracted; it must also contain all the control and data elements required to preserve its semantics. However, any data it uses that is computed by the extracted program need not be part of the co-slice; instead, it can be returned by the extracted method. These values can therefore be disconnected, making the co-slice an instance of a fine slice. Many modern languages do not allow a function or method to return more than one value. When more than one value needs to be returned to the co-slice in such languages, they can be packed into a single object. Alternatively, it is possible to selectively disconnect only one such value, making the others be recomputed by the co-slice. Another inhibitor for many languages would be the necessity of passing sequences to the extracted code and back to the co-slice. This can only be done in languages (or frameworks) that support coroutines, since it requires intertwining of the computations of the co-slice and the extracted code.

Consider a disconnected variable use u in Q. In order to determine whether it requires a single value or a sequence, we need to know whether there is a loop in P but not in Q that contains the original source of u in P and its use in Q. If there is such a loop, a sequence is necessary. We define the source of u to be the CFG node in P at which the value to be used in u is uniquely determined.

This can be a definition of the same variable, but it can also be a join in the flow at which one of several definitions is chosen. For example, a variable `x` may be set to different values in two sides of a conditional; in this case, the source of a use of `x` following the conditional is the first node that joins the flow from both assignments. Formally, we define the source of u to be the node n such that: (1) n dominates the node of u; (2) the value of the variable does not change on every path from n to the first occurrence of the node of u; and (3) n dominates every other node that fulfills conditions (1) and (2). (This will put the source of u at the same point in which a ϕ-function will be generated in the Static Single Assignment form [5].)

Theorem 4. *Given a variable use u in Q, let G_Q be the smallest strongly-connected component of Q that contains the node of u. Each edge in Q corresponds to a set of paths in P; let G_P be the sub-graph of the CFG of P that contains all the nodes and edges in P that correspond to the edges of Q. If the source of u is not in G_P, then u has a single value in the induced oracle for Q in P.*

In the example of Figure 1, the extracted code has a disconnected input `picture` in the node for `printPicture`, which is contained in a single cycle. The source of this input in the full program is the `getPicture` node, which is on the same cycle. Theorem 4 does not apply, and therefore a sequence needs to be generated for it. In contrast, the use of `end` in the predicate `i < end` is on the same cycle, but its source is the node for `Math.min`, which is not on this cycle. Therefore the theorem applies, and no sequence is necessary.

Sequences can be implemented in various ways. For simplicity of the exposition we will consider a queue, but extensions to other data structures are trivial. For those parameters that are sequences, the algorithm needs to decide where to put the call that enqueues elements in the co-slice, and where to put the call that dequeues the elements in the extracted code. This is done by locating the unique place where the data passes into Q. This place is represented by a control edge in P whose target is in Q but whose source is not, such that all control paths from the source of u to the node of u itself pass through that edge. We call this edge the *injection point for u in Q*.

Theorem 5. *The injection point for every disconnected input of an open subprogram always exists and is unique.*

Consider now the variant P' of P in which an enqueue operation immediately followed by a dequeue operation is inserted at the injection point for u. This obviously does not change the behavior of P, since the queue is always empty except between the two new operations. We now define Q' to be the subprogram of P' that, in addition to Q itself, contains the dequeue operation, and in which the input of the dequeue operation is disconnected instead of u. When performing Extract Computation on P' and Q', the enqueue operation will belong to the co-slice, while the dequeue operation will be extracted. The behavior of the

resulting program will still be the same, since the same values are enqueued and dequeued as in P'; the only difference is that now all enqueue operations precede all dequeue operations.

In order to select a location in the co-slice in which to place the call to the extracted code, it is necessary to identify a control edge in the co-slice where the call will be spliced. Such an edge is uniquely defined by its source node c, which must satisfy the following conditions (in the context of P):

- c is contained in exactly the same control cycles as $Q_{\max}$;
- c must be dominated by all sources of parameters to the extracted code;
- every path from c to any of the added enqueue operations must pass through $Q_{\min}$; and
- c dominates each node containing any input data port that is disconnected in the co-slice (and therefore expects to get its value from the extracted code).

The first condition ensures that the call will be executed the same number of times as the extracted code was in the original program. The next two conditions ensure that all parameters will be ready before the call (since passing through $Q_{\min}$ initiates a new pass through the extracted code). There may be more than one legal place for the call, in which case any can be chosen; if there is no legal place, the transformation fails (this can happen when sequences need to be passed in both directions, as mentioned above). Note that the control successor of $Q_{\max}$ satisfies the first three conditions, and the call can always be placed there unless there are results to be returned from the extracted code to the co-slice. In the example, the only valid c is the exit node.

Given a subprogram and a set of expected results, the Extract Computation algorithm proceeds as follows: (1) extract the subprogram into a separate procedure; (2) identify parameters and create sequences as necessary; (3) replace the original code by the co-slice together with a call to the extracted procedure. The Extract Computation transformation is provably correct under the assumption that all potential data flow is represented by the data dependence relation. This is relatively easy to achieve for languages such as Cobol, but may not be the case in the presence of aliasing and sharing, as in Java. In all the cases we examined as part of our evaluation (Section 4.2) this has not been an issue.

4 Discussion

4.1 Implementation

We have implemented the Extract Computation and fine-slicing algorithms on top of our plan-based slicer [3]. They are therefore language-independent, and we are using them for transformations in Cobol as well as Java in Eclipse. In particular, the example of Figure 1 is supported by our tool for both languages.

For Extract Computation, our implementation uses an extension of the Eclipse highlighting mechanism, allowing the selection of non-contiguous blocks of text. Variables or expressions that are left unmarked indicate inputs to be disconnected. In addition, we disconnect all control dependences of marked code on

unmarked code. However, as mentioned above, semantic restoration will add control structures as necessary to maintain the semantics of the extracted code.

We are investigating other applications of fine slicing. For example, clone detectors identify similar pieces of code. The obvious next step is to extract them all into one method, taking their differences into account [9]. In this case, no user input is necessary, since the parameters of the fine-slicing algorithm, and in particular, which dependences should be ignored, can be determined based on the similarities between the clones in a way that will make the extracted part identical.

Another application of fine slicing is described in the next section.

4.2 Evaluation

We conducted an initial evaluation of fine slicing in the context of a prototype system that can automatically correct certain kinds of SQL injection security vulnerabilities [4] by replacing `Statement` by `PreparedStatement` objects. A vulnerable query is constructed as a string that contains user input, and should be replaced with a query that contains question-mark placeholders for the inputs; these are later inserted into the prepared statement via method calls that sanitize the inputs if necessary. However, sometimes the query is also used elsewhere; typically, it is written to a log file. The log file should contain the actual user input; in such cases, the proposed solution is to extract the part of the code that computes the query string into a separate method, which can be called once with the actual inputs, to contstuct the log string, and again with question marks, to construct the prepared statement.

In order to automate this process, we need to determine the precise part of the code that computes the query string, with all relevant tests. The test conditions, however, should not be extracted; they can be computed once and their values passed as parameters to the two calls. This describes a fine slice that starts at the string given to the query-execution method, and ignores all data dependences on non-string values and all control dependences.

In a survey of 52 real-world projects used to test a commercial product that discovers security vulnerabilities, we found over 300 examples of the construction of SQL queries. Most of these consisted of trivial straight-line code, but 46 cases involved non-trivial control flow. In these cases, we computed a full backward slice, a fine slice according to the criteria stated above, and a data-only slice. The fine slice was computed intra-procedurally, for soundness assuming that called methods may change any field. We compared these results to the code that should really be extracted, based on manual inspection of the code.

In 21 cases, the construction of the query contained conditional parts, where the condition was the result of a method call. In all these cases, the fine slice was the same as the full slice, except that it didn't contain the method call in the conditional. In terms of lines of code, the fine slice had the same size as the full slice, although it always contained the minimal part of code that needed to be extracted. In all these cases, the data slice was too small. In practical terms, if the condition is simple and quick to compute, has no side effects, and does

not depend on any additional data, a developer may include it in the extracted code. An automatic tool, such as the one we are developing, has no information about computational complexity, and so should by default choose the minimal code, which is the fine slice.

Twenty-five cases contained more interesting phenomena, where the fine slice was strictly smaller than the full slice even in terms of lines of code. The size of the full slice was between 5 and 23 lines, with an average of 13. The fine slices were between 1 and 14 lines, with an average of 6, and always coincided with the minimal part of the code that should be extracted. The data slices were sometimes larger and sometimes smaller than the fine slices, but were correct only in three cases.

As can be seen from these results, fine slicing has proved to be the correct tool for this application. Many other applications seem to require this technology, and we will continue this evaluation on other cases as well.

4.3 Related Work

Full slices are often too large to be useful in practice. The slicing literature includes a wide range of techniques that yield collections of program statements, including forward slicing [7], chopping [8], barrier slicing [12], and thin slicing [15]. These can be used for code exploration, program understanding, change impact analysis, and bugs localization, but they are not intended to be executable, and do not preserve the semantics of the selected fragment in the original program. In particular, they do not add the required control structures. We believe that all these techniques for finding interesting collections of statements could benefit from the added meaning given by semantic restoration, not only for use in program transformations, where executability with semantics preservation is a must, but also in assisting program understanding and related programming tasks such as debugging, testing, and verification.

Tucking [13] extracts the slice of an arbitrary selection of seed statements by focusing on some single-entry-single-exit region of the control flow graph that includes all the selected statements. They refer to this limited-scoped slice as a *wedge*. A tuck transformation adds to the identified region a call to the extracted wedge and removes from it statements that are not included in the full slice starting from all statements outside the wedge. Our computation of the co-slice by starting a fine slice from all nodes not in the extracted code is similar in this respect.

Using *block-based slicing*, Maruyama [14] extracts a slice associated with a single variable in the scope of a given block into a new method. The algorithm disconnects all data dependences on the chosen variable; this could lead to incorrect results in some cases. Tsantalis and Chatzigeorgiou [16] extended this work in several ways, including rejecting the transformation in such problematic cases. They still use the same framework of limiting the slice to a block. Fine slicing provides much finer control over slice boundaries.

A procedure-extraction algorithm by Komondoor and Horwitz [10] considers all permutations of selected and surrounding statements. Their following paper

[11] improves on that algorithm by reducing the complexity and allowing some duplication of conditionals and jumps but not of assignments or loops. Instead of backward slicing, this algorithm completes control structures but only some of the data. If a statement in a loop is selected, all the loop is added.

Sliding [6] computes the slice of selected variables from the end of a selected fragment of code, and composes the slice and its complement in a sequence. The complement can be thought of as a fine slice of all non-selected variables, ignoring all dependences of final uses of the selected variables. Our Extract Computation refactoring is more general by possibly ignoring other dependences, and by allowing more flexible placement of the slice. The concept of a final use of a variable can also help choosing which dependences to ignore when extracting a computation.

None of these approaches support passing sequences of values to what we call an oracle variable.

4.4 Future Work

This work is part of a long-term research project focusing on advanced enterprise refactoring tools, aiming to assist both in daily software development and in legacy modernization. The Extract Computation refactoring is a crucial building block in this endeavor. It will be used to enhance the automation for complex code-motion refactorings in order to support enterprise transformations such as the move to MVC [1,2]. As the prototype matures, it will be possible to evaluate to what extent such enterprise transformations can be automated.

We intend to make a number of improvements to the underlying analysis. Most important are interprocedural analysis and some form of pointer analysis. These will also support interprocedural transformations. The semantic restoration algorithm is useful on its own in order to make any subprogram executable. Based on our preliminary investigation we believe that an interprocedural extension of this algorithm is straightforward. It only requires pointer analysis in the presence of polymorphism, in order to compute the most accurate calling chain to be restored.

Acknowledgments. We are grateful to Mati Shomrat for his help with the implementation, and to Moti Nisenson for the name fine slicing.

References

1. Abadi, A., Ettinger, R., Feldman, Y.A.: Re-approaching the refactoring Rubicon. In: Second Workshop on Refactoring Tools (October 2008)
2. Abadi, A., Ettinger, R., Feldman, Y.A.: Fine slicing for advanced method extraction. In: Proc. Third Workshop on Refactoring Tools (October 2009)
3. Abadi, A., Ettinger, R., Feldman, Y.A.: Improving slice accuracy by compression of data and control flow paths. In: Proc. 7th Joint Mtg. European Software Engineering Conf. (ESEC) and ACM Symp. Foundations of Software Engineering (FSE) (August 2009)

4. Abadi, A., Feldman, Y.A., Shomrat, M.: Code-motion for API migration: Fixing SQL injection vulnerabilities in Java. In: Proc. Fourth Workshop on Refactoring Tools (May 2011)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Programming Languages and Systems 13(4), 451–490 (1991)
6. Ettinger, R.: Refactoring via Program Slicing and Sliding. Ph.D. thesis, University of Oxford, Oxford, UK (2006)
7. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12(1), 26–60 (1990)
8. Jackson, D., Rollins, E.J.: A new model of program dependences for reverse engineering. In: Proc. 2nd ACM Symp. Foundations of Software Engineering (FSE), pp. 2–10 (1994)
9. Komondoor, R.: Automated Duplicated-Code Detection and Procedure Extraction. Ph.D. thesis, University of Wisconsin–Madison (2003)
10. Komondoor, R., Horwitz, S.: Semantics-preserving procedure extraction. In: Proc. 27th ACM Symp. on Principles of Programming Languages (POPL), pp. 155–169 (2000)
11. Komondoor, R., Horwitz, S.: Effective automatic procedure extraction. In: Proc. 11th Int'l Workshop on Program Comprehension (2003)
12. Krinke, J.: Barrier slicing and chopping. In: Proc. 3rd IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM) (September 2003)
13. Lakhotia, A., Deprez, J.C.: Restructuring programs by tucking statements into functions. In: Harman, M., Gallagher, K. (eds.) Special Issue on Program Slicing, Information and Software Technology, vol. 40, pp. 677–689. Elsevier (1998)
14. Maruyama, K.: Automated method-extraction refactoring by using block-based slicing. In: Proc. Symp. Software Reusability, pp. 31–40 (2001)
15. Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: Proc. Conf. Programming Lang. Design and Implementation (PLDI), pp. 112–122 (June 2007)
16. Tsantalis, N., Chatzigeorgiou, A.: Identification of Extract Method refactoring opportunities for the decomposition of methods. J. Systems and Software 84(10), 1757–1782 (2011)
17. Weiser, M.: Program slicing. IEEE Trans. Software Engineering SE-10(4) (1984)