Automatically Adapting Source Code to Document Provenance

Simon Miles

Department of Informatics, King's College London, UK

Abstract. Being able to ask questions about the provenance of some data requires documentation on each influence on that data's existence and content. Much software exists, and is being developed, for which there is no provenance-awareness, i.e. at best, the data it outputs can be connected to its inputs, but with no record of intermediate processing. Further, where some record of processing does exist, e.g. as logs, it is not in a form easily connected with that of other processes. We would like to enable compiled software to record useful documentation without requiring prior manual adaptation. In this paper, we present an approach to adapting source code from its original form without manual manipulation, to record information on data provenance during execution.

1 Introduction

Many systems have been developed where the processing performed is documented during execution. The documentation allows us to answer questions about the processes which led to a data item being produced, i.e. its *provenance*. The documentation commonly contains copies of intermediate data items, otherwise discarded by the completion of a process, and causal dependencies between data items. In some cases, recording is performed *automatically* and *transparently*, as a side-effect of the execution, without either the author or user of a process being involved in what is recorded or how.

Such automatic, transparent recording has been built into workflow systems [**?**], and operating environments in which user actions are performed, e.g. the Provenance-Aware Storage System [**?**] or ES3 [**?**]. In the former, transparent recording means documenting the connection between data on them being inputs and outputs to the same workflow step. In the latter, OS (or higher) events are intercepted to document how an executed process reads and writes files.

Whether processes are enacted as pre-scripted workflows or ad-hoc user actions, the component steps are compiled from some *source code* and the intermediate data created within compiled components or the details of how outputs depend on inputs may be as important as intermediate data or dependencies at the workflow/OS level. In some cases, we want a record of what occurred during the execution of compiled code. To some extent, this can be provided by logging, but here there is no interoperability with the wider execution setting: we wish to know not only the list of events occurring between the executable's

initiation and termination, but also that the executable's inputs were themselves provided as part of a larger, distributed workflow. Such interoperability is a goal of the Open Provenance Model (OPM) [**?**], which documents processes as causal graphs allowing independently produced graphs to be combined.

Braun et al. [**?**] considered issues inherent in achieving automatic collection of provenance data. One consideration is that the *granularity* of provenance information a user is interested in is generally coarser/higher-level than an automatic collection mechanism records. It is important that some parts of an execution could and, sometimes can only, remain relatively *opaque*, described in coarse-grained terms. Another, related, consideration is that, for storage and privacy reasons, it may be undesirable to automatically record all that could be. A way to resolve this is to allow, but not require, *configuration* of what will be recorded, thus retaining transparency as far as it does not have a negative effect.

In more theoretical work, Souilah et al. [**?**] provided a formalism for languages expressible as asynchronous π-calculus, whereby the provenance of values exchanged across channels (and of the channels themselves) would be maintained throughout execution. In this work, the values are automatically augmented with their provenance as metadata, and thus propagated through the system. Processes can make decisions (accepting a value or not) by filtering on static patterns within the provenance data, which is expressed as a list of communication events. Buneman et al. [**?**] examined how the semantics of database query and update languages implied the provenance of data within the databases, and so could automatically be augmented with actions to record how the data was transformed. The provenance was expressed as the propagation of colours, denoting data remaining the same in value or in kind, as the database was transformed. In both the latter papers, a correctness property of the provenance, with regards to what actually occurred in the system, was articulated and proven.

In this paper, we describe a preliminary approach, *SourceSource*, whereby source code is automatically adapted to document its processing during execution. While we argue the approach is generally applicable to procedural languages, our preliminary work is applied to a case study Java program. To enable configuration by the developer, where desired, the code retains its original form as far as possible in the adapted form, with recording statements inserted in the same language. Configuration is aided by treating the code as a set of components, some amenable to adaptation and some not (opaque), and allowing the developer to decide if a component should not be adapted to record its execution except at a coarse-grained level (merely connecting component inputs to outputs). The documentation is recorded as an OPM causal graph, allowing the program's execution to be connected with the preceding processes producing the program's inputs and succeeding processes consuming its outputs.

2 Overview and Case Study

Following OPM, the *provenance* of some data is the *processes* and *artifacts* which ultimately cause it to exist, and causal relations between them. An artifact is

a constant data item and a process is the execution of some procedure, taking artifacts as input (*used* relation) and producing artifacts as output (*wasGeneratedBy* relation). An artifact may be generated by one process then later used by another, indirectly connecting the two processes. A set of OPM artifacts, processes and relations between them forms an *OPM graph*. An OPM graph may document the provenance of multiple artifacts, and we say that the graph is a set of *process documentation* from which the provenance of individual artifacts can be extracted by querying. Finally, each artifact, process and relation can be annotated with multiple *annotations*, each having a type and value.

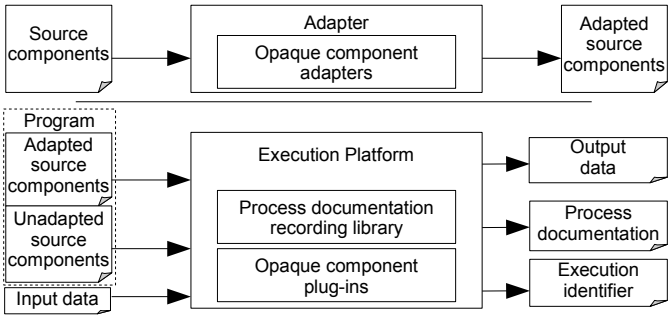

Fig. 1. Adaptation of source code (top) and then its execution (bottom) (top)

Fig. **??** (top) depicts SourceSource architecture. The *adapter* component takes source code and produces an adapted version. A program comprises one or more *source components* for which the code is adaptable, e.g. class files in Java, plus one or more *opaque components* for which code is not available, e.g. databases, third-party libraries. The user can choose which components are adapted. The adapted version is augmented to interleave execution with calls to a recording library to document execution, making use of *opaque component adapters* triggered for statements where an opaque component is used. When the code is executed (bottom of figure) on its standard platform, the execution of the adapted components is automatically documented. For an opaque component, the relevant plug-in is invoked to document the execution (as far as is possible) and connect it to the source execution which called the component. The recording library outputs process documentation and an identifier for the execution.

We take as our case study the workflow used in the third Provenance Challenge[1], an analysis of astronomy data from the Pan-STARRS project [**?**], implemented in Java. The aim of our approach is to automatically adapt code so that OPM documentation of its execution is recorded, without prior modification of the code to suit our approach. Therefore, a key fact about the case

[1] `http://twiki.ipaw.info/bin/view/Challenge/`

study is that it was developed *independently* from our work, with no knowledge that the SourceSource approach would be applied. In the provenance challenge, queries were performed over the process documentation to demonstrate its efficacy. For example, one query, inspired by program slicing [**?**], asked 'Which operation executions were strictly necessary for the Image table to contain a particular (non-computed) value?'. We refer readers to the challenge website for more details.

3 Process Documentation

We first describe the process documentation produced in our approach, then in Section **??** explain how SourceSource adapts code to record this. SourceSource primarily documents statements being executed (processes in OPM) and variables having particular values (OPM artifacts). A variable may take on multiple values during execution, so the recording library holds a mapping from each variable name to the artifact denoting its *most recent* value. When a variable has a new value, this is recorded as a new artifact by the adapted code and the *most recent* mapping is updated. When a variable is used in a statement, the most recent artifact for that variable is found, and connected by a causal (used) relation to the process representing the statement's execution. For example, in Fig. **??** (top), the first statement assigns a value to a variable `FileEntry`, then the second depends on that value, and this dependency holds even if there were other statements between the two shown.

```
for (LoadAppLogic.CSVFileEntry FileEntry : ReadCSVReadyFileOutput) {
  boolean IsExistsCSVFileOutput = LoadAppLogic.IsExistsCSVFile (FileEntry);

LoadWorkflow_main_Statement5
for (LoadAppLogic.CSVFileEntry FileEntry : ReadCSVReadyFileOutput) {
  LoadWorkflow_main_Declaration6
  boolean IsExistsCSVFileOutput = LoadAppLogic.IsExistsCSVFile (FileEntry);

for (LoadAppLogic.CSVFileEntry FileEntry : ReadCSVReadyFileOutput) {
  Recorder.process ("LoadWorkflow_main_Statement5");
  Recorder.variable ("LoadWorkflow_main_FileEntry", "LoadWorkflow_main_Statement5", FileEntry);
  Recorder.pass ("IsExistsCSVFile", 0, "LoadWorkflow_main_FileEntry");
  Recorder.push ();
  boolean IsExistsCSVFileOutput = LoadAppLogic.IsExistsCSVFile (FileEntry);
  Recorder.pop ();
  Recorder.process ("LoadWorkflow_main_Declaration6");
  Recorder.variable ("LoadWorkflow_main_IsExistsCSVFileOutput", "LoadWorkflow_main_Declaration6",
                     IsExistsCSVFileOutput);
  Recorder.generated ("LoadWorkflow_main_IsExistsCSVFileOutput", "Assigned Value In",
                      "LoadWorkflow_main_Declaration6");
  Recorder.used ("LoadWorkflow_main_Declaration6", "LoadWorkflow_main_FileEntry",
                 "Used In Expression");
```

Fig. 2. A snippet of code from the case study unadapted (top), with naming annotations (middle), and after adaptation (bottom)

3.1 Identifiers and Querying

Regardless of query language, a user querying for the provenance of data, maybe long after the process which produced it took place, will only have certain information available. Such a user cannot be expected to have available identifiers which are not part of the source code, input data, or outputs of adaptation or execution, and just because an identifier is somewhere in the process documentation we cannot assume the user knows what it identifies.

We can decompose a provenance query into: identifying the start item of which to find the provenance; expressing what in the potentially large set of documentation connected to the start item is relevant for the query; and how the relevant documentation is post-processed to answer the query. All stages require identifiers (for input, output and intermediate data) to be known. We take the following approach to identification in SourceSource (see Chapman and Jagadish on the general problems of identifying intermediate data [**?**]).

- An execution of a statement is identified by: the statement's scope identifiers (e.g. package and method names in Java), a unique *statement identifier* generated by SourceSource, and the count of which *iteration* of this statement this execution denotes (how many times the statement has been executed).
- A variable value is identified by: the statement execution where the value is assigned/used, the variable's scope and its name.
- Each program execution is identified by a generated *execution identifier*.

The identifiers above are annotated to the relevant artifacts/processes in the OPM graph. The execution identifier is also the filename of the serialised graph, allowing a user to connect the execution with its documentation. In combination, the above ensure there is a unique way to identify each documented artifact/process across executions, and identifiers can be known to querying users through being connected to the artifact/process either in the original source code, the adapted source code, or the execution output.

In the case study, each Java statement is given a name scoped by its class, e.g. `LoadWorkflow_main_Statement5` (in class LoadWorkflow, in method main, the 5th statement). These can be used to query for the provenance of the iterations of executing the statements. A tool is provided to see what names statements are given to aid those building queries. A snippet of the output this tool produces is shown in Figure **??** (middle), where each statement is preceded by its identifier. The opaque plug-ins must identify processes and artifacts appropriately for their components. For the case study database plug-in, described below, each database entry is given an identifier comprised of the table name and the primary key fields of the entry, e.g. `table=P2DETECTION, objID=113191992826421637.`

3.2 Granularity and Procedure Calls

The provenance of a data item can be expressed in different ways, suitable for different purposes. In particular, a description of its provenance can be expressed at a coarser or finer *granularity* of detail. OPM allows for multiple granularities

of documentation to be demarcated by *account* identifiers. At a coarse granularity, the execution of a procedure call can be described as a black-box process which produces outputs given inputs, implying a possible causal connection between the outputs and inputs. At a fine granularity, the execution of a procedure call can be described as the caller's arguments being used as inputs to a succession of processes which comprise the procedure executed, ultimately resulting in outputs returned to the caller. SourceSource always records the coarse-grained account for a procedure call in an adapted procedure, and will record the fine-grained account where the called procedure is also adapted. The two accounts are connnected, by the identifiers of the call inputs and outputs being common to both accounts, and by an OPM refinement relationship.

Each procedure's execution is documented as a separate OPM sub-graph, with the final graph produced from the program's execution being the union of those sub-graphs. In Figure **??**, we illustrate a snippet of the OPM graph produced by executing the adapted case study source, corresponding to the part shown in Figure **??**. There are three accounts: one for the execution of the main method (`FineGrained1`), one for the invocation of the `IsExistsCSVFile` method (`CoarseGrained2`), and one for the execution of that method (`FineGrained2`).

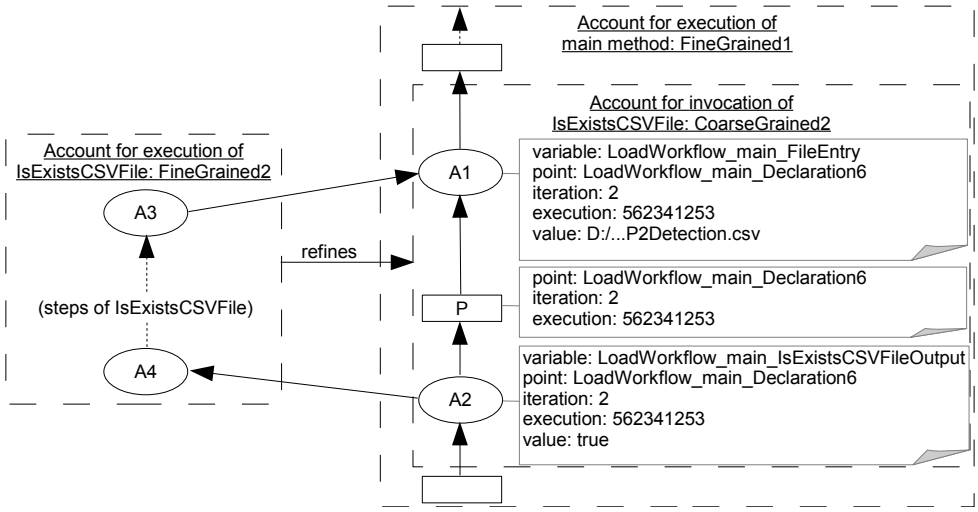

Fig. 3. Fragment of OPM graph produced in case study (ovals denote artifacts, solid rectangles denote processes, relations are arrows pointing from effect to cause, and key-value lists are annotations, dashed rectangles demarcate accounts)

In account `CoarseGrained2` we see an artifact A1 for the `FileEntry` variable passed as argument to the invocation process, P, and A2 for the value assigned to variable `IsExistsCSVFileOutput`. In account `FineGrained2`, the same values are assigned to variables local to method `IsExistsCSVFile`, and are different

artifacts derived from (and identical in value to) A1 and A2. Each artifact and process is annotated with its identifier (Section **??**), plus variable values for artifacts. If the developer chose not to adapt the source component containing IsExistsCSVFile, the left-hand part of the graph would be excluded, without disconnecting the graph on the right-hand side.

4 Adaptation

Adaptation consists of three stages: *explicate* to transform code to enable addition of recording statements; *identify* to determine unique (within a single execution) identifiers for every occurrence which will be documented as an artifact or process; and *augment* to insert recording statements interleaved with execution. To perform adaptations in all three stages, we use TXL [**?**], a tool to transform source code from one form to another. A TXL rule takes a subtree of a given form, and transforms it into another subtree following the same grammar.

4.1 Explicate and Identify Stages

The first stage of adaptation is to, as minimally as possible, add structure to the code to enable recording steps to be inserted. In Java, this means that the body of `if` and similar control statements are represented as blocks (`if (C) then {X}`) rather than single statements (`if (C) then X`).

As discussed above, to be able to query the process documentation after recording, the entities to be referred to must be identifiable. This has two implications: *(i)* entities which have no natural identifier in the source code must be given one, *(ii)* the identifier to use on recording should be determined for each entity prior to augmenting with recording statements using those identifiers. In the first case, the primary entities in question are the statements, as they do not by default have a name by which they can be referred by a querier.

The first phase of the *identify* stage is to go through and annotate each statement with a name unique within its method, e.g. Statement5. TXL allows *attributes* to be inserted into the parse tree which will not be apparent in the transformed output, but can be used by other rules, and so the names are prepended to statements as such. In the second phase, we use attributes to provide global identifiers to code entities (variables, methods, statements). We ensure names are unique across the execution by constructing them from their scopes, e.g. a method's local variable is named by the package name, class name, method name, and variable name. As described in Section **??**, we provide a tool to enable the identifiers of each source code line to be seen. A snippet of the output is shown in Fig. **??** (middle).

4.2 Augment Stage

In the *augment* stage, for each occurrence of a process/artifact, and for each causal relation between them, a recording statement is inserted into the code.

Processes could include method calls, expression evaluations, or variable assignments. Currently, we augment at the statement level. In Fig. **??** top and bottom, we show a snippet of the case study code before and after augmentation. There is a loop across the elements of a collection, whose body's first line is the assignment of the result of a method call to a new variable (new each time round the loop). In augmenting that code snippet, our TXL scripts insert the following recording statements (other operations on the recording API document receipt of parameters, return of values from methods etc.)

1. Each loop iteration expression is a process, so a statement is inserted at the start of the loop body to document this process, named `..._Statement5` in the *identify* stage. On execution, this recording statement will insert a process node into the OPM graph.
2. We document the state of the loop variable, `..._FileEntry`, which will insert an artifact into the OPM graph with the variable's new value, and put the value the *most recent* mapping in memory.
3. The next statement in the original source includes a method call, so we need to keep track of the arguments passed to connect them with the parameters inside the called method. We store this as a tuple: the method being called, the index of the argument and the most recent value of the variable passed.
4. Each method invocation is documented in a separate account. The `push` method continues any subsequent recording in a new account, pushing the current one onto a stack. When the invocation completes, `pop` returns to the original account being used and creates a refinement relationship between the call (in a new coarse-grained account) and the invoked method accounts.
5. A process node is recorded for the assignment statement, and a generated artifact for the newly assigned variable.
6. We record relations, `used` and `generated`, documenting the artifacts (variable values) were used and generated by the assignment statement.

In many cases, source code which could be adapted using SourceSource will make calls to libraries, databases or other components for which the SourceSource approach cannot apply. Where this occurs, the call to the component in the source code can be adapted to invoke a *plug-in* which handles recording process documentation for components of that type (before and/or after the call, as appropriate). For the case study, we developed and used one plug-in, very much tailored to the case study, for the database holding the experiment results.

5 Conclusions

Where distributed processes include a compiled tool which does not record any documentation about its processing, the provenance of those processes' results will be more limited, will exclude some potentially relevant intermediate data items, and may be disconnected (it may not be apparent where the tool's outputs depend on its inputs). Making a tool provenance-aware manually can be

expensive, so we would rather that the tool could, without manual modification, automatically recorded documentation during execution. The solution we present in this paper is to automatically adapt the tool's source code to record documentation in OPM. The approach is particularly applicable where the tool's developer should have control over the recording, e.g. to manage volume, to protect privacy, to remove irrelevant details. By inserting recording statements into the code in the same language, we require nothing of the developer but make it easier for them to configure recording afterwards.

The work described here is preliminary, so while everything achieved in SourceSource can be applied to any Java program, only those code features essential for completing the case study have been tested. While the principles of only minimally changing the code structure and using the same source language are adhered to, there are undoubtably improvements possible in how recording statements are inserted, e.g. to ensure low overhead costs. Checking and improving performance overhead requires a larger case study: both the original and adapted case study code execute trivially quickly.