Domain-Specific Program Profiling and its Application to Attribute Grammars and Term Rewriting[☆]

Anthony M. Sloane[a,*], Matthew Roberts[a]

[a]*Department of Computing, Macquarie University, Sydney, Australia*

Abstract

We present a method for profiling programs that are written using domain-specific languages. Instead of reporting execution in terms of implementation details as in most existing profilers, our method operates at the level of the problem domain. Program execution generates a stream of events that summarises the execution in terms of domain concepts and operations. The events enable us to construct a hierarchical model of the execution. A flexible reporting system summarises the execution along developer-chosen model dimensions. The result is a flexible way for a developer to explore the execution of their program without requiring any knowledge of the domain-specific language implementation.

These ideas are embodied in a new profiling library called *dsprofile* that is independent of the problem domain so it has no specific knowledge of the data and operations that are being profiled. We illustrate the utility of *dsprofile* by using it to profile programs that are written using our Kiama language processing library. Specifically, we instrument Kiama's attribute grammar and term rewriting domain-specific languages to use *dsprofile* to generate events that report on attribute evaluation and rewrite rule application. Examples of typical language processing tasks show how domain-specific profiling can help to diagnose problems in Kiama-based programs without the developer needing to know anything about how Kiama is implemented.

Keywords: program profiling, attribute grammars, term rewriting, Kiama, Scala

[☆]Nikki Lesley, Suzana Andova and Dominic Verity provided helpful feedback on this work. Some of the project was performed while the first author was a guest of Mark van den Brand's group at TU Eindhoven. The authors thank the anonymous reviewers for their useful suggestions which greatly improved the paper.

[*]Corresponding author

Email addresses: Anthony.Sloane@mq.edu.au (Anthony M. Sloane), Matthew.Roberts@mq.edu.au (Matthew Roberts)

1. Introduction

All modern computer code is executed at a different level of abstraction than it is defined. It is the job of a compiler or interpreter to translate between the two levels of abstraction. Profiling tools for general-purpose languages routinely bridge this abstraction gap by presenting the execution of a program in terms of the program code, not in terms of the lower-level execution platform. For example, gprof [1] and many tools inspired by it base their reports on the functions that appear in the source code of the program. A developer can see which functions are called, how much execution time they consume, and how control flows between them. A gprof profile does not report on lower-level details such as machine instructions, stack frames or register usage.

Executable domain-specific languages (DSLs) [2] are no exception to this requirement for profilers to bridge an abstraction gap. In fact, the burden on DSL profilers is greater because DSLs are based on higher-level abstractions than general-purpose languages. Just as we want gprof to present function-level information and not details of the implementation of those functions, a DSL profiler should report at the level of domain-specific abstractions and not at the DSL implementation level. For example, the implementation language for the DSLs in this paper is Scala [3], so object creations and method calls are performed as a DSL program executes. However, objects and methods are not concepts of the DSLs, so a gprof-style profiler for Scala would not be appropriate for DSL profiling since it would require a DSL user to understand how the DSL is implemented. Instead, we aim to make it possible to construct profilers that collect and present information that we can reasonably expect a DSL user to be familiar with.

To this end we have developed a new general method for building DSL profilers with the following aspects that generalise those used in gprof-style profilers:

1. *A hierarchical event-based model of program execution.* The events in the model represent instances of domain-specific program operations being applied to domain-specific data. The inclusion relation encoded by the hierarchy represents the extent to which operations use other operations, generalising gprof-style tools that present execution as a control-flow based hierarchy of function calls.
2. *Domain-specific event dimensions.* Each model event has a set of values associated with it that distinguish it from other events of the same type. The values belong to dimensions that are chosen by the profiler author to be appropriate for the domain. Dimensions generalise the fixed properties such as function name that are used by a gprof-style profiler.
3. *A simple query language and reporting framework.* Developers query the execution of a program by nominating one or more dimensions of interest. A report generator summarises the events of the execution in terms of the nominated dimensions. Events with the same value for a nominated dimension are aggregated in the report. This aggregation generalises that

performed by a gprof-style profiler to aggregate information pertaining to each function in a program.

All of these aspects are independent of the problem domain that is addressed by the DSL, so the approach is applicable to any executable DSL. We have implemented them in our new *dsprofile* library. To keep our examples concrete, we use DSLs from the software language engineering domain in the examples in the rest of the paper. Specifically, we describe how we have used dsprofile to add profiling support to our Kiama language processing library [4]. Examples are chosen from typical language processing tasks that use Kiama's *attribute grammar* and *strategy-based term rewriting* DSLs.

The rest of this introduction briefly introduces the attribute grammar and term rewriting DSLs by way of simple programs and profiles of those programs. Section 2 begins the paper proper with a detailed description of the execution model, of how event information is aggregated to produce reports, of our implementation in dsprofile, and how dsprofile is used in Kiama. Sections 3 and 4 use examples based on a version of Java to show how profiling of attribute grammars and term rewriting, respectively, can yield valuable information about program execution when we implement typical language processing tasks. The paper concludes with a review of related work (Section 5) and an examination of directions for future extensions (Section 6).

1.1. Attribute grammars

Attribute grammars promote a view of tree decoration based on declarative equations defined on context-free grammar productions [5]. An attribute is defined by equations that specify its value at a node N as a function of constant values, the values of other attributes of N, and the values of attributes of nodes that are reachable from N. Provided that sufficient equations are defined to cover any context in which the node can occur, we obtain a declarative specification of an algorithm that can be used to compute the attribute of any such node. This approach to computation on structures has been shown to be extremely powerful. Recent applications that use attribute grammars heavily are XML integrity validation [6], protocol normalization [7], Java compilation [8], image processing [9], and genotype-phenotype mapping [10].

Consider the following simple abstract tree structure for constant expressions defined in Scala:

```
abstract class Exp
case class Num(i : Int) extends Exp
case class Add(l : Exp, r : Exp) extends Exp
case class Mul(l : Exp, r : Exp) extends Exp
```

Using this AST definition, the expression `3 + 4 * 5` can be represented by the value `Add (Num (3), Mul (Num (4), Num (5)))`.

The value of an expression can be defined as an attribute in Kiama as follows:

```
val value : Exp => Int =
  attr {
    case Num(i)    => i
    case Add(l, r) => value(l) + value(r)
    case Mul(l, r) => value(l) * value(r)
  }
```

Kiama's `attr` operation takes care of caching attribute values, detecting dependency cycles, etc. (For full details see our paper on Kiama's attribute grammar library [11].)

Similarly, an attribute that returns true if and and only if an expression has a zero value can be defined as follows:

```
val iszero : Exp => Boolean =
  attr {
    case n => value(n) == 0
  }
```

A single attribute definition such as for `value` or `iszero` is usually fairly simple, but in realistic applications the effect of an attribute is intricately intertwined with that of many other attributes. The value of an attribute A is ultimately determined not just by A's equations, but by the equations of any attribute on which A's equations transitively depend. Thus, the computations that cooperate to compute a value of A are potentially dispersed throughout the attribute grammar. This dispersal means that it is non-trivial to determine what an attribute value is or even how an attribute is calculated just by looking at the equations.

Powerful extensions of the original attribute grammar formalism make this problem worse. Equations in the original conception of attribute grammars can only refer to attributes of symbols that occur in the context-free grammar production on which the equation is defined. In higher-order and reference attribute grammars the value of an attribute can itself be a reference to a node upon which attributes can be evaluated [12, 13]. This extension is particularly useful when defining context-sensitive properties such as name binding or programmer-defined operator precedence. This power comes at a price, however, because it means that more parts of the grammar are in play when we are trying to understand a particular attribute and how it is computed.

The situation is complicated even more by the fact that many modern attribute grammar systems use a dynamically-scheduled evaluation strategy, which precludes accurate static analysis. Some classes of attribute grammars submit to static dependence analysis that enables evaluation strategies to be computed in advance of running the evaluator [14]. One can imagine tools based on this static analysis that would assist with understanding the evaluation process. More recently, however, attribute grammar tools and libraries have mostly used an approach where the evaluation strategy is determined at run-time [11, 15, 16, 17]. This dynamic approach admits more grammars than

does a static approach, some algorithms are easier to express, and features such as higher-order and reference attributes are easier to support. However, a dynamically-scheduled approach also means that an accurate static analysis is not possible. Instead, dynamic analysis is necessary since evaluation will be influenced by the attribute values, which will in turn be influenced by the input.

Our profiling system enables data to be collected as the program executes and presents it in a simple tabular format. For example, suppose that we are interested in the execution of the following code where `exp` is the tree for `3 + 4 * 5`:

```
println (iszero (exp))
println (value (exp))
```

Running this code under the control of the profiler described in this paper collects information about every attribute evaluation and their dynamic dependencies. We can use the query "name cached" to ask for a profile using as the primary dimension the name of the attribute and as the secondary dimension the cached dimension that records whether a computed attribute value has been obtained from the attribute's cache or not. The profile for this run includes the following table that shows aggregated information for the name dimension:

```
By name:

 Total Total  Self  Self  Desc  Desc Count Count
    ms     %    ms     %    ms     %           %
     0  97.3     0  23.4     0  73.8     1  14.3  iszero
     0  76.6     0  76.6     0   0.0     6  85.7  value
```

The table summarises the run-time by apportioning it to the attributes. Each row first shows the total time taken by the evaluation of the attribute as an absolute time and a percentage. The next four columns show the portions allocated to the equations of the attribute itself (`Self`) and of the attributes that those equations used (`Desc` for "descendants"). The final two columns show the number of times each attribute was evaluated.

We can see that the `iszero` attribute is evaluated once and most of its time is taken up by attributes that it uses. The `value` attribute is evaluated six times which is what we would expect (once for each of the five nodes in the tree to help print the `iszero` value of the root, and once again at the root to print the value itself. Thus, this profile allows us to confirm that the program is behaving as we would expect.

The profile also contains a table showing the secondary cached dimension aggregated by the first dimension values:

```
By cached for iszero:

 Total Total  Self  Self  Desc  Desc Count Count
    ms     %    ms     %    ms     %           %
```

```
     0  23.4     0  23.4     0   0.0     1  14.3  false

By cached for value:

 Total Total  Self  Self  Desc  Desc Count Count
    ms     %    ms     %    ms     %           %
     0  73.8     0  73.8     0   0.0     5  71.4  false
     0   2.7     0   2.7     0   0.0     1  14.3  true
```

From this table we can see that the `iszero` value was not obtained from its cache, but that one of the `value` calls was since the value at the root has already been calculated while computing the `iszero` value.

These sorts of profiles reveal many details about how an attribute performs and bridge the gap between the way the DSL is implemented and the user's understanding of their program. We present larger attribute grammar examples from the language engineering domain in Section 3.

1.2. Strategy-based rewriting

Term rewriting is concerned with tree transformation [18]. In this paper we focus on strategy-based rewriting in the style of the Stratego language [19, 20], but the ideas should transfer easily to other forms of rewriting. Strategies attempt to rewrite a tree and either succeed and produce a new tree, or fail. The simplest form of strategy is a rewrite rule that describes how to match certain tree structures and rewrite them into new structures. A rewrite rule fails if the match fails. More complex strategies are built using operators. For example, a choice operator allows the success or failure of a strategy to guide future rewrites. Generic traversal operators allow transformations of whole trees to be written in a concise way that only mentions relevant structure. Term rewriting has been widely used [21] and recent application areas include integrated development environments [22] and speech recognition [23].

We can easily define rewriting transformations on the arithmetic expression AST defined in Section 1.1. For example, we can write a simplifier for expressions that replaces an expression with a simplified form if addition by zero or multiplication by zero or one is present:

```
val simplify =
  rule {
    case Add (Num (0), e) => e
    case Add (e, Num (0)) => e
    case Mul (Num (1), e) => e
    case Mul (e, Num (1)) => e
    case Mul (Num (0), _) => Num (0)
    case Mul (_, Num (0)) => Num (0)
    case n                => n
  }
```

Kiama's `rule` operation defines a rewrite rule based on the user-supplied cases. The last case ensures that a node is left alone if no optimisation is possible. (For more details on Kiama's rewriting library see Stratego [19, 20] and our comparison between Kiama and Stratego [24].)

We can use the simplifier in a more complex strategy that applies it across a whole AST using Kiama's `bottomup` library function.

```
val optimise = bottomup (simplify)
```

`bottomup` takes care of details such as traversing the AST, freeing the user from programming the traversal.

By themselves, rewrite rules such as `simplifier` and strategies such as `optimise` are usually fairly simple. However, the transformation process described by groups of cooperating strategies can be very complex. For example, a generic traversal like `bottomup` traverses the tree in a specific way which will be influenced by the behaviour of its argument strategy. It is often hard to understand from the strategies themselves exactly where in the tree rewrites will happen or why they don't happen as expected at a particular tree node. In fact, the errors that occur in term rewriting are so particular that there has been specific work to catalogue them [25] and on static analysis to identify them [26].

Understanding where the strategies are applied and whether they succeed or fail is a big help when debugging rewrite systems. Just as it collects information about attribute evaluations, our profiling system monitors rewrite rule and strategy applications. For example, we obtain the following profile when we apply the optimiser to the expression `0 + (1 * (5 + 4) + 0)` and ask for the "name" dimension:

```
By name:

  Total Total  Self  Self  Desc  Desc Count Count
     ms     %    ms     %    ms     %           %
      2  99.7     0   3.2     2  96.5     1   2.4  optimise
      2  96.3     1  52.8     1  43.5    14  33.3  all
      2  91.1     1  39.7     1  51.4    13  31.0  bottomup
      0   4.3     0   4.3     0   0.0    14  33.3  simplify
```

As for the attribute profiles, the rewriting profile shows the times and execution counts for each rule or strategy. (`all` is a library operation that is used in the definition of `bottomup`.) We apply fourteen simplifications since the rewrite is applied to every node in the tree: five integer leaves, five expressions containing those leaves, and four internal expression nodes. As could be expected for this simple example, we spend most of the time traversing the tree, not performing simplifications.

These sorts of profiles reveal much about how rewrite rules and strategies apply and interact. As for the attribute grammar DSL, they bridge the gap between the way the rewriting DSL is implemented and the user's understanding of their program. We present larger rewriting examples from the language engineering domain in Section 4.

1.3. An alternative: HPROF

Of course, profiling tools are available for the platform on which the Kiama is built (i.e., Java and Scala). Unfortunately, these profilers are not capable of effectively illuminating the execution of code defined in the Kiama DSLs because they report their results at the wrong abstraction level.

To provide direct evidence of this assertion, we used the built-in Java profiler HPROF to profile the optimisation from the previous section. Since the computation runs very quickly we repeated it one hundred times so that there was something to profile. A very small part of the HPROF profile is shown in Figure 1. The full profile runs to over eight hundred lines because there is one line for each *Java* method called. In the figure we see some Kiama methods used in rewriting plus some Scala collection methods.

```
0.01% 91.58%    2828 337738 org.kiama.rewriting.Strategy$$anonfun$1.<init>
0.01% 91.59%    4060 301904 java.lang.Character.toLowerCase
0.01% 91.60%    4961 308045 scala.collection.AbstractTraversable.genericBuilder
0.01% 91.61%    4863 308043 scala.collection.immutable.List$.newBuilder
0.01% 91.62%     930 313536 scala.reflect.internal.Scopes$Scope...
0.01% 91.63%    1414 337830 org.kiama.rewriting.RewriterCore$$anonfun$all$1...
0.01% 91.64%    4242 337851 scala.collection.mutable.ListBuffer.$plus$plus$eq
0.01% 91.66%    4242 337852 scala.collection.mutable.ListBuffer.result
```

Figure 1: A very small part of the profile generated by the Java run-time when running the optimisation of Section 1.2 one hundred times. The query string "hprof=cpu=times" was used to generate the profile.

It is evident that the profile in Figure 1 is not of much use to a DSL user compared to the profiles that we presented earlier. If that user has knowledge of the Kiama implementation and of the way in which Scala programs are compiled to the Java virtual machine it is possible for them to be able to find the calls that correspond to the application of their rewriting strategy. However, that knowledge is very unlikely to be available so this sort of profile is almost always useless. In contrast, the profiles produced by Kiama using dsprofile allow the user to explore the execution of the rewrite at an appropriately high level of abstraction.

1.4. Previous work and available software

This paper is an extended and revised version of one that appeared in the proceedings of the 2012 International Conference on Software Language Engineering [27]. All of the sections have been extensively revised and expanded, but the material relating to rewriting systems is new, most notably Section 4. All examples now use Kiama notation.

Code and documentation for our implementation of the profiling framework can be found at `http://bitbucket.org/inkytonik/dsprofile`. This paper is based on version 0.3 of `dsprofile`.

Kiama can be obtained from `http://kiama.googlecode.com`. This paper is based on version 1.5.2 of Kiama. Since the SLE paper the profiling code for

attribute grammars has been added to the main Kiama distribution. Support for profiling circular attributes and rewriting has been added.

2. Domain-Specific Program Profiling

Profiles such as those shown in the introduction reveal a lot about the execution of a DSL program in a form that is easy to absorb. Varying the choice of dimension in a query enables us to tailor the profiles to the particular investigation that we are carrying out. In this section we describe in detail how the profiles are constructed, particularly how the program execution is modelled, how reports are produced, and what must be done in Kiama to use the profiler. Sections 3 and 4 present examples of using the Kiama profiler in realistic software language engineering applications.

Our approach is to instrument programs to generate *domain-specific events* while they run. The events are grouped to form a *record-based model* of the execution in domain-specific terms. Reports are generated from the model by summarising records along developer-specified *dimensions* using a simple query language. In Kiama we are focusing on profiling for attribute grammars and term rewriting, but profilers can be built for any domain by varying the events that are generated and the dimensions that are used to summarise the execution.

Section 2.1 describes the data collection method and the record-based model of execution. Section 2.2 describes the query language and explains how the model is used to produce reports. The implementation of the dsprofile library is discussed in Section 2.3, Section 2.4 describes how we are using dsprofile in Kiama, and the performance of the implementation is analysed in Section 2.5.

2.1. Data collection and execution modelling

The data collection approach is based on a simple event model. We distinguish between `Start` events that signal the beginning of some program activity and `Finish` events that signal the end of an activity. We assume that the program can be modified so that `Start` and `Finish` events will be generated at appropriate times. Usually these events would be generated by the DSL implementation, but they can be generated from user code as well.

Each event captures the event kind (`Start` or `Finish`), the time at which it occurred, the domain-specific type of the event, and a collection of arbitrary data items associated with the event instance. Each data item is tagged with a unique dimension. The dimensions that an event has at generation time are its *intrinsic dimensions*, to differentiate them from *derived dimensions* that are calculated later.

For example, in the attribute grammar case, a single *attribute evaluated* event type is sufficient. We abbreviate this event type by `AttrEval` in the profiles. A `Start` instance of this event type is generated just before the evaluation of an attribute begins, and a corresponding `Finish` instance is generated just after evaluation of an attribute ends. Attribute evaluation events have the following intrinsic dimensions:

- *attribute*: the attribute that was evaluated,
- *subject*: the node at which the attribute was evaluated,
- *parameter*: the value of the attribute's parameter (if any),
- *value*: the value that was calculated by the attribute's equations, and
- *cached*: whether the value was calculated or came from the attribute's cache.

Generation of the `Start` event produces an *event id* which must be used in the generation of the `Finish` event as a sanity check that events are properly nested. The `Finish` event can have dimensions that were not present in the `Start` event. For example, for attribute evaluation the *value* and *cached* dimensions are present only in the `Finish` event since that information is available only after the evaluation has been completed.

After execution is complete, we collect the events to create a list of *profile records* that describe the execution. When we see a `Finish` event we match it with the corresponding `Start` event. Each matching `Start-Finish` pair is represented by a single profile record. A record contains the event type, the time taken between the occurrence of the `Start` event and the occurrence of the `Finish` event, and all of the intrinsic dimensions from the two events.

We also require that the events are generated in a last-in-first-out manner so that we can automatically construct a hierarchical model of execution. In other words, if we see a `Finish` event, it must be the case that the most recently seen `Start` event that does not yet have a corresponding `Finish` event is the one that corresponds to the new `Finish` event. If this condition holds, we can regard the records that are created between a `Start` event and its corresponding `Finish` event as the *descendants* of the new profile record. This hierarchical relationship is used to derive dimensions that relate records to each other; for example, to summarise attribute dependencies (Section 3.6).

To make this description concrete, consider the execution of the attribute evaluator from the introduction, calculating the `iszero` attribute and then the `value` attribute at the root of the tree representing `3 + 4 * 5`. Among the events generated by this evaluator will be some that document the evaluation of the `iszero` and `value` attributes. A possible execution results in the events shown in the top table of Figure 2. This trace excerpt describes the seven attribute evaluations. The first `Start` event marks the start at time step four of the evaluation of the `iszero` attribute at the `Add` node. That evaluation requires an evaluation of the `value` attribute at the same node, which in turn demands the `value` attribute at the `Num(3)` node, and so on. The final attribute evaluation at time step 11 asks for a value that was already computed at time step 9 so the computed value can be obtained from the attribute's cache.

Seven profile records will be created to represent this execution, one for each attribute evaluation (bottom table of Figure 2). For example, record four tells us that the evaluation of `value` at the `Mul` node took a total of five time units

Kind	*When*	*Attribute*	*Subject*	*Value*	*Cached*
`Start`	4	`iszero`	`Add`		
`Start`	4	`value`	`Add`		
`Start`	4	`value`	`Num(3)`		
`Finish`	5	`value`	`Num(3)`	3	false
`Start`	5	`value`	`Mul`		
`Start`	5	`value`	`Num(4)`		
`Finish`	6	`value`	`Num(4)`	4	false
`Start`	6	`value`	`Num(5)`		
`Finish`	7	`value`	`Num(5)`	5	false
`Finish`	8	`value`	`Mul`	20	false
`Finish`	9	`value`	`Add`	23	false
`Finish`	10	`iszero`	`Add`	false	false
`Start`	11	`value`	`Add`		
`Finish`	12	`value`	`Add`	23	true

Record	*Time*	*Attribute*	*Subject*	*Value*	*Cached*	*Descs*
1	1	`value`	`Num(3)`	3	false	
2	1	`value`	`Num(4)`	4	false	
3	1	`value`	`Num(5)`	5	false	
4	3	`value`	`Mul`	20	false	2, 3
5	5	`value`	`Add`	23	false	1, 4
6	6	`iszero`	`Add`	false	false	5
7	1	`value`	`Add`	23	true	

Figure 2: Start and Finish events encoding attribute evaluations on the tree representing `3 * 4 + 5` (top) and the profile records that summarise the attribute evaluations encoded by the events (bottom). Note: the attribute evaluations in the bottom table are ordered by completion time, not start time.

and required the evaluations represented by records two and three. Record two in turn took one time unit and did not require any other evaluations.

2.2. Query language and report generation

The record list produced by the data collection process is a domain-specific model of the execution. The user can query this model using a simple query language whose queries consist simply of an ordered sequence of dimension names. For example, to produce the first report in the introduction we used the query "name cached", referring to the name and cached dimensions.

The report generation process proceeds by considering the records one-by-one and allocating them to *report buckets* according to their query dimension values. All of the records with the same query dimension value end up in the same bucket and their execution time is accumulated. The descendant information allows us to allocate the elapsed time to either the attribute evaluation represented by a record (self) or to the other evaluations demanded by that evaluation (descendants). When the buckets have been collected, a table is printed wherein each row summarises a bucket and the rows are sorted in decreasing order of execution time. For example, for the one-dimensional query "name" the sole table will contain one row for each unique name dimension value.

Multi-dimensional reports are produced by an analogous process. We first report on the basis of the first query dimension. Then each bucket from the first dimension table is further summarised according to the second query dimension. For example, for the "name cached" query, the first table will report on the name dimension, while the remaining tables will report on the cached value for each discrete name value. The process continues until there are no more query dimensions to consider.

The query dimensions can be provided at compile time or a program can enter a "profiler shell" in which a set of dimensions can be given interactively and the shell will respond with the reports for those dimensions. This design allows a data-driven exploration of the records collected during profiling.

Our student Nathan Seal has also developed a graphical profile viewer that lets the user select dimensions interactively from a graphical user interface. Profiles can be displayed in various graphical forms such as histograms or pie-charts.

2.3. Implementation

The profiles presented in this paper were collected using our *dsprofile* library which implements our profiling approach as a library in the Scala language. Profiles can be generated from code written in any Java Virtual Machine language but specific support is provided for Scala and Java. Section 2.4 describes how Kiama uses the library to generate profiles.

An important decision we made in the implementation was to not encode the available dimensions and the types of their values into the profiling library. A dimension is represented by a string and a dimension value can be any value. Including more type information would enable a greater level of safety in the

event generation and recording code. However, it would tie the library to particular events and their dimensions. Adding new ones would require updating the interfaces or a more complex event representation with generic access to typed dimension values. Our approach is simple and extremely flexible. Instrumentation code is one line at each event generation site. New events and dimensions can be added without recompiling the library.

A dsprofile component of around 400 lines of code implements event capture, record representation, and report generation. The current implementation stores profile records as instances of a custom class. A generic format such as XML or JSON could easily be used instead and might be beneficial if the data was to be exported. As it stands, the event data is only used internally by the library, so this generality is not needed. We have not used any form of compression to reduce the space needed to store the profile records, since it has not been necessary for the test cases we have tried. If space usage becomes a problem, an on-the-fly approach could be the solution, where aggregation is performed as events are generated rather than at the end of the execution.

Events are time-stamped using the `java.lang.System.nanoTime` method that has nanosecond precision. As in all profiling systems, the measured times vary from run to run depending on the machine load, but the relative times are stable. Precise nanosecond times are unlikely to be very useful since they present too much detail, so the profiler reports times in millisecond units.

Profiles that use intrinsic dimensions can be generated directly from the profile records. Derived dimensions can be added by overriding the default implementation of library method `dimValue` that looks up dimension values. The method is given the dimension name and a reference to the profile record. The default implementation simply looks up the name in the record's dimension collection. An overriding implementation can return any value it likes. (Section 2.4 gives an example of how to code a derived dimension.)

The display of aggregated values can also be customised. By default, the profiler uses the standard Java `toString` method to obtain a string representation of a value. That implementation can be replaced by arbitrary code. For example, we could display subject trees using a pretty-printer instead of using the default representation. Since these sorts of values can take up a significant amount of space, they are unlikely to fit in the profile report tables. The report writer automatically detects when the value strings will not fit. Each such value is allocated a footnote number which is used in the table. The actual value string is printed below the table.

2.4. Instrumenting Kiama

dsprofile is independent of the problem domain and the DSL implementations within that domain. As a concrete example of using dsprofile with sophisticated DSLs, we instrumented our Kiama language processing library [4, 11], which is written in Scala, to use dsprofile to collect information about attribute evaluations and rewrites.

For example, to evaluate normal attributes we use the code shown in the top part of Figure 3 which resides in the cached attribute class. The `apply` method

```
def apply (t : T) : U = {
  val id = start (Seq (
               "event" -> "AttrEval", "subject" -> t,
               "attribute" -> this, "parameter" -> None,
               "circular" -> false
           ))
  resetIfRequested ()
  memo.get (t) match {
    case None     =>
      reportCycle (t)
    case Some (u) =>
      finish (id, Seq ("value" -> u, "cached" -> true))
      u
    case _ => // null
      memo.put (t, None)
      val u = f (t)
      memo.put (t, Some (u))
      finish (id, Seq ("value" -> u, "cached" -> false))
      u
  }
}
```

```
 def apply (r : Any) : Option[Any] = {
   val i = start (Seq (
               "event" -> "StratEval", "strategy" -> this,
               "subject" -> r
           ))
   val result = body (r)
   finish (i, Seq ("result" -> result))
   result
 }
```

Figure 3: Kiama code that evaluates attributes (top) and applies rewriting strategies (bottom) including instrumentation code to generate profiling events using the `start` and `finish` calls.

is called when an attribute is evaluated at a node `t` of generic type `T`, returning a value of generic type `U`. The code manages the attribute's cache (`memo`) and reports a cycle if this attribute is evaluated again while we are computing its value. Otherwise, the value is either obtained from the cache if the attribute has been evaluated before, or is calculated, stored in the cache and returned.

The instrumentation in Figure 3 comprises the call to `start` and the two calls to `finish`, one for each of two cache lookup outcomes. The `AttrEval` event dimensions and their values are passed in a sequence using Scala's arrow tuple notation; e.g., `"attribute" -> this` is a tuple comprising the dimension name and the attribute instance. The `id` returned by `start` must be passed to the corresponding `finish` call to enable the library to check that events are properly nested.

Events for term rewriting are generated similarly. The bottom part of Figure 3 shows the `apply` method of the rewrite strategy class. This method is called when a strategy is applied to a subject term `r`. The method returns an optional value, representing either failure of the application or success producing a new subject term. As for the attribute case, `start` and `finish` are called to produce an event for each strategy evaluation.

Overall, the event generation code is a small addition to the Kiama code. In the attribute grammar there are a total of seventeen calls to `start` and `finish` to deal with the different kinds of attribute evaluation. The rewriting instrumentation is much simpler and requires only one call to each of the methods, since rewriting strategy evaluation is localised in one place.

Since Kiama's implementation contains the instrumentation, users of Kiama need to only add a call to the dsprofile library to run their code under profiler control. For example, to evaluate the `iszero` attribute from the introduction on the expression `e` we write:

```
profile (iszero (e), Seq ("name", "cached"))
```

The first argument to `profile` is the computation that is to be run with profiling turned on. The second argument is a sequence containing the query dimensions; when the computation is finished, the profiler will print a report for these dimensions. If no query dimensions are provided in the `profile` call the profiler will enter a mode where queries can be entered interactively. We can profile a rewriting computation with a similarly simple call. For example, the profile of the optimisation from the introduction is produced by the following call, where `e` is the expression being optimised:

```
profile (optimise (e), Seq ("name"))
```

Calling another dsprofile method causes the profiler to print an event trace, filtering events by a user-supplied predicate. We omit further discussion of trace printing since it is not germane to the main topic of the paper.

Around 300 lines of support code in Kiama implements derived dimensions for events by overriding the dsprofile `dimValue` method which is responsible for returning the value of a particular dimension from a profiling record. For

```
def dimValue (record : Record, dim : Dimension) : Value =
  dim match {
    case "subjectHash" =>
      checkFor (record, dim, "", "subject") {
        case s => s.##
      }
  }
```

Figure 4: Kiama implementation of a derived dimension for the hash code of the subject node of an attribute evaluation or rewriting operation.

example, Figure 4 shows how we implement a "subjectHash" derived dimension which is useful for distinguishing between two tree node instances that have the same printed representation. The derived dimension is based on the "subject" intrinsic dimension used by attribute evaluations and rewriting to report the node on which the evaluation or rewriting is being performed. `checkFor` is provided by dsprofile to check for the presence of a particular dimension in a record and, if it is present, provide its value to an arbitrary computation. In this case, the computation simply calls the Scala hash code method `##` and returns its result. While this and other derived dimensions are packaged with Kiama, it is also possible for DSL users to define derived dimensions in the same way, thereby allowing them to view their execution in problem-specific terms.

A DSL implementation that is an embedding in a general-purpose language faces the problem of obtaining access to general-purpose names for use in DSL messages. For example, in Kiama we need to gain access to the name of an attribute (e.g., "iszero" in the introduction) so that we can use it if we need to report a dependency cycle while evaluating an attribute. Similar requirements apply to Kiama's term rewriting rules. Earlier versions of Kiama required the user to specify these names as a string when they defined the attribute or rule. Current versions of Kiama remove this requirement by gaining access to the name from the Scala abstract syntax tree using Scala's macro facility [28]. The Kiama profiling support uses this name access to implement the "name" derived dimension that is used in our examples.

2.5. Performance

The performance of a profiling library is not important in production, but can make a difference to the efficiency of the development process. dsprofile contains some measures to remove overhead from the data that it gathers. Most notably, the time overhead of generating events is accumulated as the events are generated. When the profile is prepared the total overhead is apportioned evenly to each event and deducted from the event's total execution time. We use this approach since it spreads responsibility for the overhead across all events, rather than to just the event that was being generated at the time when the

overhead was incurred. Therefore, we avoid allocating responsibility for non-trivial overhead such as garbage collection to a single event.

To explore the performance of our implementation, we conducted some experiments using an attribute grammar that is much bigger than the expression language from the introduction or the PicoJava example we use in Sections 3 and 4.

The Oberon-0 example was developed for the tool challenge associated with the 2011 Workshop on Language Descriptions, Tools and Applications (LDTA). Oberon-0 is the imperative language subset of the Oberon family of languages and was originally described by Wirth [29]. The challenge compiler parses and analyses Oberon-0 programs, then translates correct ones into equivalent pretty-printed C code. The Kiama Oberon-0 specification can be found in the Kiama distribution at `kiama/src/org/kiama/example/oberon0`. It contains the definitions of fourteen attributes whose definitions use sixty-one separate cases to define equations.

Our test case was to compile fifty-four Oberon-0 test programs in a single run of the compiler. None of these programs is very large. Their total representation in the Oberon-0 compiler comprises around six thousand AST nodes. The compiler performs more than thirty-two thousand attribute evaluations to perform name and type analysis on these trees, so it is a serious test of the profiling system.

Running the Oberon-0 compiler on this test with profiling completely disabled takes about five to six seconds of elapsed time. Adding the event generation code to the library, but still with no report generation, doesn't make a difference that is noticeable in the usual variation when running from the command line. We also modified the Oberon-0 compiler to collect the run-time for just the core compiler driver, thereby removing the time for other operations such as class loading. We ran all of the Oberon-0 tests in a single run as before, repeating the run ten times initially to warm up the virtual machine. Then we ran twenty-four tests, discarded both the slowest two and the fastest two results to remove any outliers, and averaged over the remaining twenty measurements. The results showed that the event generation by itself slows the core of the compiler down by a factor of about 1.4. While this is a significant difference, the command-line experiment shows that the slowdown is swamped by the time taken by other operations performed by the program. Thus, we believe that the instrumentation is practical for gathering data from large test runs.

We also investigated the time taken to produce profile reports. Producing a profile for the attribute dimension increases the run-time from around eight seconds with profiling turned on but no report generation, to about twelve seconds with report generation as well. Adding a second subject dimension increases the total time to over twenty-two seconds. Most of this time is taken by printing the many tree fragments, which illustrates that report generation time is highly dependent on the chosen dimensions. We have not performed any optimisation of the core of the report generator so it is likely that some improvement could be obtained. Nevertheless, our experiments show that the current performance is practical for typical interactive uses during development.

```
{
  class A {
    int y;
    AA a;
    y = a.x;
    class AA {
      int x;
    }
    class BB extends AA {
      BB b;
      b.y = b.x;
    }
  }
}
```

Figure 5: A PicoJava program containing a class and two nested classes.

3. Profiling Attribute Evaluation

We now consider in more detail the use of our profiling approach to examine the execution of non-trivial programs. This section examines name analysis, a typical software language engineering task, and shows how profiling can help us to understand the operation of a name analysis attribute grammar implementation.

3.1. PicoJava

The attribution we consider performs name analysis for a Java subset called PicoJava. This attribution was originally written as an illustration of reference attributes in the JastAdd system [30]. The Kiama distribution contains a fairly direct translation of the JastAdd attribute grammar. We do not intend to describe Kiama's attribute grammar DSL [11] in full, but to explain enough to understand the example. Although the Kiama attribution DSL provides some syntactic conveniences, in essence attributes in Kiama are normal functions that memoise their results.

PicoJava contains declarations and uses of Java-like classes and fields, but omits most of the expression, statement and method complexity of Java. Figure 5 shows a PicoJava program consisting of a block with a declaration of class `A`. Class `A` contains two nested classes: `AA` and `AA`'s sub-class `BB`. Statements are limited to simple assignments between named objects which are either fields of the current object or qualified accesses to fields of other objects.

The problem that is solved by the attribute grammar is to check the uses of all identifiers. For example, the uses of `x` in the `a.x` and `b.x` expressions are legal because of the declaration of the `x` field in `AA` and the inheritance

```
class Program (Block : Block)
class Block (BlockStmts : Seq[BlockStmt])
class BlockStmt

class Decl (Name : String) extends BlockStmt

class TypeDecl (Name : String) extends Decl (Name)
class ClassDecl (Name : String, Superclass : Option[IdnUse],
                 Body : Block) extends TypeDecl (Name)
class PrimitiveDecl (Name : String) extends TypeDecl (Name)
class UnknownDecl (Name : String) extends TypeDecl (Name)

class VarDecl (Type : Access, Name : String) extends Decl (Name)

class Stmt extends BlockStmt
class AssignStmt (Var : Access, Value : Exp) extends Stmt
class WhileStmt (Cond : Exp, Body : Stmt) extends Stmt

class Exp
class Access extends Exp
class IdnUse (Name : String) extends Access
class Use (Name : String) extends IdnUse (Name)
class Dot (ObjRef : Access, IdnUse : IdnUse) extends Access
class BooleanLiteral (Value : String) extends Exp
```

```
Program (
  Block (List (
    ClassDecl ("A", None (),
      Block (List (
        VarDecl (Use ("int"), "y"),
        VarDecl (Use ("AA"), "a"),
        AssignStmt (
          Use ("y"),
          Dot (Use ("a"), Use ("x"))),
        ClassDecl ("AA", None (),
          Block (List (
            VarDecl (Use ("int"), "x")))),
        ClassDecl ("BB", Some (Use ("AA")),
          Block (List (
            VarDecl (Use ("BB"), "b"),
            AssignStmt (
              Dot (Use ("b"), Use ("y")),
              Dot (Use ("b"), Use ("x"))))))))))))
```

Figure 6: A class model for the abstract syntax of the PicoJava language (top) and the abstract syntax tree for the program in Figure 5 (bottom).

relationship between `AA` and `BB`. However, the use of `y` in `b.y` is illegal since `BB` and `AA` declare no `y` field, even though there is a `y` field in the enclosing class `A`.

The attribute grammar operates on an abstract syntax tree representation of the PicoJava program. Our version uses the same AST structure as the original JastAdd example. A class model for this structure is shown in the top part of Figure 6. A program contains a sequence of block statements which are either declarations or statements. Types and variables can be declared. Types are user-defined classes, primitive types or unknown types that are used for error cases. Statements are restricted to assignments and while loops. Expressions only have forms that are relevant for name analysis: basic identifier use and "dot style" reference to a member of an object.

The bottom part of Figure 6 shows the AST for the program in Figure 5. Note that the superclass component of a user-defined class is represented by an optional value which is either `Some(c)`, if the superclass is `c`, or `None`, if there is no declared superclass.

3.2. Name analysis for PicoJava

The attribute grammar that implements name analysis for PicoJava defines one main attribute `decl` whose value is the declaration corresponding to a particular access of a name. `decl` is a reference attribute that refers to the actual `VarDecl` or `ClassDecl` node in the tree. `decl` is a *synthesized attribute*, meaning that it is defined for all productions that define the structure of memory accesses (for which we define the class `Access` from which all such productions inherit).

```
val decl : Access => Decl =
  attr {
    case u : Use    => u->lookup (u.Name)
    case Dot (_, n) => n->decl
  }
```

We first declare the type of the attribute. The `decl` attribute is defined on `Access` nodes and its type is `Decl`, the common superclass of `VarDecl` and `ClassDecl`. The arrow `=>` is Scala's function type constructor.

PicoJava accesses come in two varieties: a direct use of a single identifier (`Use`) or a field reference with respect to a qualified object access (`Dot`). A `Dot` contains an access defining the accessed object and an identifier use that names the member that is being accessed.

In the attribute grammar the two varieties of access are selected by two cases in the pattern matching function that is passed to Kiama's `attr` method. The cases are the attribute equations in a Kiama-based attribute grammar. Normal Scala pattern matching is used in the cases. For example, the first case uses a type pattern that matches any `Use` node and binds it to the name `u`. The second case matches any `Dot` node and binds the name `n` to the second (`Use`) component. The first component is ignored by a wildcard pattern.

Each case in the attribute definition has a right-hand side that describes how to compute the attribute for an argument that is selected by that case. References to members of a node are written using a "dot" notation while references to attributes are written using an "arrow" notation. For example, in the first case `u.Name` refers to the string in the `Use` node. The expression `n->decl` is asking for the value of the `decl` attribute for the node bound to `n`; the same expression could be written in normal functional notation as `decl(n)`.

In order to describe how to compute an attribute value, an attribute equation can refer to any symbols of the associated production to obtain data values from attributes or intrinsic properties. Expressions can use any facility of the host environment to compute with these values. The overall effect of the `decl` equations is that the relevant declaration is determined by evaluating the `lookup` attribute at the rightmost identifier use in the access. For example, when evaluating `decl` for the expression `a.b.c`, which is represented by the tree `Dot(Use("a"),Dot(Use("b"),Use("c")))`, we will evaluate `decl` for `b.c`, `decl` for `c` and, finally, `lookup("c")` at the `Use("c")` node.

3.3. Name lookup

The `lookup` attribute searches to find the declaration that matches a particular name. The value of `n->lookup(s)` is a reference to the node that represents the declaration of `s` when viewed from the scope of `n`, or a reference to a value of type `UnknownDecl` if no such node can be found.

```
val lookup : String => Attributable => Decl = ...
```

The first argument is the name that is being searched for; we will refer to it as `name` in the code below. The second argument is the node at which we are searching. We indicate that the attribute can be evaluated at any node using the type `Attributable`. Usually all node types in the abstract syntax tree definition inherit from `Attributable` to obtain members that give generic access to the tree structure. For example, in the definitions below we use the `parent` field that is a reference from a node to its immediate parent node (if any).

`lookup` is a *parameterised attribute*, since it depends on the name being sought. It is also an *inherited attribute* since its value will be determined by the context surrounding the node where it is evaluated, not by the inner structure of that node.

One case is when `lookup` is evaluated at a `Use` node.

```
case i : Use =>
  i.parent match {
    case Dot (a, i2) if i eq i2 =>
      a->decl->tipe->remoteLookup (name)
    case p =>
      p->lookup (name)
  }
```

If the use is part of a qualified access via a `Dot` we must lookup the name in the type of the access qualifier. This search is achieved by getting the declaration of the qualifier (`a->decl`), getting that declaration's type (`->tipe`)[1], and performing a remote lookup for the name in that type (`->remoteLookup (name)`). `tipe` returns a reference to the node representing the class type from a declaration. `remoteLookup` looks for a name in a class type from the perspective of a client of that type. We omit the definitions of the `tipe` and `remoteLookup` attributes since those details are not necessary for our discussion.

The second sub-case for `Use` corresponds to the use of an unqualified name, so we just continue the search at the parent node to search in the current scope. This action is also used at any node that is not handled by a more specialised case.

The remaining cases for `lookup` propagate requests coming from `Use` nodes to the appropriate parts of the tree and trigger searches in blocks and superclasses. First, we consider the case where the search reaches a block statement. We use the node's parent reference to check if the node is at the top level of a block. If so, we search locally in the block first. If that search fails, or we are not yet at the top of a block, we move the search to the parent node.

```
case s : BlockStmt =>
  s.parent match {
    case b : Block =>
      if (isUnknown (b->localLookup (name)))
         b->lookup (name)
      else
         b->localLookup (name)
    case p =>
      p->lookup (name)
  }
```

If we are searching at a block (as opposed to inside it), we distinguish two cases. First, the block might be the one for the program as a whole, in which case we search locally there. Second, the block might be in a class declaration, in which case we look for a superclass. If there is a superclass we search remotely in that class and return a declaration found there. If there is no superclass, or there is one but it has no declaration of the name we are seeking, we move the search to the parent node.

```
case b : Block =>
  b.parent match {
    case p : Program =>
      p->localLookup (name)
    case c : ClassDecl =>
      if ((c->superClass != null) &&
```

[1]The identifier `tipe` is used because `type` is a Scala reserved word.

```
        !isUnknown (c->superClass->remoteLookup (name))))
      c->superClass->remoteLookup (name)
    else
      c->lookup (name)
}
```

3.4. Understanding PicoJava name analysis

Name analysis attribution for PicoJava is a canonical example of reference attribute grammars [30]. The equations are not lengthy, but their operation is still quite hard to understand. Some of the difficulty is due to the inherent complexity of the problem being solved. The tasks of name and type analysis for a language like PicoJava are intertwined. For example, to look up a name in the body of a class, we may need to search the superclass type, which involves performing name analysis on the name that appears in the superclass position of the class declaration, and so on, while avoiding problems such as cycles in the inheritance chain.

Given this inherent complexity, it is somewhat surprising that the definitions are not longer than they are. The main reason for their brevity is the power of the dynamically-scheduled attribute grammar formalism to abstract away tree traversal details. When we are writing our equations, we can reason about how declarations, local and remote lookups, and types relate to each other, without having to work out a particular tree traversal that evaluates the attributes in the correct order. The attributes are evaluated when their values are first demanded and caching means that we don't need to worry about which particular use of an attribute asks for it first. In contrast, a solution based on tree traversals implemented by visitors, for example, would have to explicitly plan which attributes should be evaluated at which time. Developing such a plan is a non-trivial task for this problem.

It is not possible to completely ignore tree traversal. When we are developing and debugging the equations, we need help to understand how they function. It is not enough to just look at each equation by itself, since the effect is achieved by a combination of many equations. Having some information about what happens at run-time can reveal much about how the attribute grammar works. As a simple example, if we knew that our name analyser never examined the super class of a class when processing the program in Figure 5, we would know that the equations were not correctly implementing our intent.

3.5. Profiling PicoJava name analysis

In this case, the interesting program behaviour is in terms of attributes, their values, and so on. Profiles produced as we described in Section 2 enable us to see which attributes are being evaluated and how they are consuming time in the name analysis computation.

The simplest profile we can imagine is one that shows us which attributes are evaluated during a run (query "event name" and look at `AttrEval` table). The top of Figure 7 shows this profile for the PicoJava name analyser as it processed

```
   113 ms total time
    31 ms profiled time (28.1%)
   231 profile records

By name for AttrEval:

 Total Total  Self  Self  Desc  Desc Count Count
    ms     %    ms     %    ms     %           %
    28  90.3     1   5.4    27  84.9    28  12.1  decl
    27  84.9     7  22.2    20  62.7    32  13.9  lookup
    14  46.8    12  37.8     2   9.0    19   8.2  localLookup
     5  17.4     1   4.6     4  12.8    18   7.8  unknownDecl
     4  13.6     2   9.1     1   4.5     7   3.0  remoteLookup
     3  10.8     2   6.4     1   4.4    20   8.7  tipe
```

```
By type for AttrEval and lookup:

 Total Total  Self  Self  Desc  Desc Count Count
    ms     %    ms     %    ms     %           %
    27  84.9     2   8.3    24  76.6    12   5.2  Use
    17  53.6     1   3.8    15  49.8     4   1.7  VarDecl
     4  15.1     1   3.4     3  11.7     5   2.2  Block
     3   9.7     1   3.3     2   6.5     4   1.7  ClassDecl
     1   6.1     0   2.2     1   4.0     4   1.7  AssignStmt
     1   5.7     0   1.3     1   4.5     3   1.3  Dot
```

Figure 7: Extracts of profiles produced when the PicoJava name analyser processes the program in Figure 5: query "event name" (top) and query "event name type" (bottom).

the program in Figure 5.[2] The first part of the profile gives the total run-time and the time that is accounted for by the profiled attributes. 206 attribute instances were evaluated in this run.

The profile table shows that the `decl` attribute and the attributes it uses consume the vast majority of the time, closely followed by `lookup`, and further back, `localLookup`. The profile reveals a number of areas where further investigation might be warranted. The `localLookup` attribute consumes almost a half of the time which seems excessive. We might investigate whether replacing a linear search by a hashed lookup would improve performance. Also, the `unknownDecl` attribute is used to return a special object to represent the case where a declaration cannot be found. It is worrying that computing this special

[2]Elapsed time is collected in nanosecond units, but is presented in the profiles as milliseconds, so there may be some rounding errors.

object consumes seventeen percent of the time.

The top of Figure 7 shows just a single dimension: the attribute that was evaluated. Our profiles can summarise execution across more than one dimension to reveal more detail. For example, we might want to know the types of the nodes at which the `lookup` attribute was evaluated. The bottom profile in Figure 7 shows the `lookup` part of a multi-dimensional profile using the attribute and node type dimensions (query "event name type"). In this table the rows summarise a particular combination of the `lookup` attribute and node type. For example, the first line summarises the cases where the `lookup` attribute was evaluated at `Use` nodes. Since the tree in Figure 5 contains three `Dot` nodes, it is comforting to see from the sixth line of the table that we looked up names three times at such nodes.

3.6. Derived dimensions

As discussed in Sections 2.3 and 2.4 it is straightforward for the DSL developer or DSL user to add new dimensions that are derived from the ones that are intrinsically part of profiling events. For example, when writing attribute grammar code it is sometimes useful to be able to see where attributes are being evaluated in the tree. Kiama provides a derived dimension called "location" to aid with this analysis. It summarises the location of the subject node of an attribute evaluation within the tree as "Root", "Inner" or "Leaf" depending on which of these categories it falls into. The dimension implementation makes use of node properties that are provided by Kiama such as the `parent` property used in the name analysis attributes in Section 3.2. A typical use case might involve a query such as "name cached location" to see where in the tree attribute caches are being used.

A more complex derived dimension is one that summarises the dynamic dependencies between attribute occurrences. Kiama's *depends-on* derived dimension aggregates attribute evaluations according to the attributes on which they directly depend. It can be used to check that the pattern of attribution that is being performed is as desired. The depends-on dimension is implemented in twelve lines of code. A related *dependencies* dimension considers all transitive dependencies and generates visualisations for display by GraphViz. The implementation of this dimension is more complex and requires around seventy lines.

Since derived dimensions are implemented by arbitrary code their processing can be arbitrarily complex. There is no real limit on the extent to which derived dimensions can be tailored to support profiling within a particular domain. In particular, they can be used by a DSL user to extend the basic facilities of the profiler provided by the DSL implementer. Of particular interest are problem-specific extensions that enable the DSL user to construct custom profiles that suit exactly what they are trying to achieve in their program.

3.7. More complex analysis via attribute profiling

PicoJava name analysis uses plain attributes and parameterised ones. Our profiling approach extends to *circular attributes* where the evaluation is iterated

until a fixed point is reached.

As well as looking at single attributes, we can also use profiles to compare different designs for a collection of attributes. For example, the PicoJava name analyser uses parameterised attributes to search for the declaration information to bring it to the uses where it is needed. An alternative approach to name analysis attribution is to propagate an environment value around the tree which collects declaration information as it goes. When the environment reaches a use of an identifier, it can be accessed directly to check that use.

Kiama supports this form of attribution via a *chain*, which is inspired by a similar construct in the LIGA attribute grammar system [31]. A chain abstracts a pattern of attribution that threads a value in a depth-first left-to-right fashion throughout a tree. The idea is that the system provides the default threading behaviour and the attribute grammar writer can customise the equations at various places in the tree to update the chain value. Kiama chains are implemented by a pair of attributes: one to calculate the value of the chain that comes in to a node from its parent, and one to calculate the value that goes back out of the sub-tree to the parent. The developer can provide functions to transform the incoming value as it heads into a sub-tree or as it leaves the sub-tree.

We can use profiling to compare these two general approaches to attribute evaluation for name analysis. The conclusions will depend on the exact patterns of identifier use that are present in typical programs. Generating profiles for different test cases will reveal the overall cost of each approach and enable us to compare the individual evaluations that make up either propagation or lookup.

4. Profiling Term Rewriting

We now consider the use of dsprofile for a second domain: strategic term rewriting. We show that the scheme described in Section 2 provides information that helps us to understand the dynamic behaviour of rewrites, just as it did for the attribute grammar domain.

4.1. Obfuscating PicoJava programs

Suppose that we want to obfuscate PicoJava programs by renaming variables and classes. Each declared variable and class should be allocated a unique number. Uses of variable and class identifiers should be replaced by the letter `"n"` followed by the number of the entity to which the identifier is bound. Predefined identifiers such as `int` should not be renamed. We assume that all identifiers refer to a declared entity since this transformation would normally be run on legal programs. Figure 8 shows an example where the program on the left is transformed into the obfuscated form on the right.

In general, the same identifier can be used in a program to refer to more than one entity. Therefore, this program transformation requires name analysis information to ensure that each identifier is replaced by the new name of the correct entity. For example, in the program on the left of Figure 8 there are two `avar` variables. The transformation needs to take into account that these are different variables, in this case resulting in the names `n1` and `n7`, respectively.

```
{                                    {
  class ALongClassName {               class n0 {
    int avar;                            int n1;
    int bvar;                            int n2;
    class NestedClass {                  class n3 {
      int item;                            int n4;
      avar = item;                         n1 = n4;
    }                                    }
    NestedClass object;                  n3 n5;
    object.item = bvar;                  n5.n4 = n2;
  }                                    }
  class AnotherClassName {             class n6 {
    int avar;                            int n7;
    ALongClassName object;               n0 n8;
    avar = object.bvar;                  n7 = n8.n2;
  }                                    }
}                                    }
```

Figure 8: A PicoJava program (left) and its obfuscated form (right).

4.2. The obfuscation transformation

Declarations and uses may be mixed arbitrarily in a PicoJava program. It is not necessarily the case that a declaration of an identifier occurs textually before a use of that identifier. Therefore, the obfuscation transformation consists of the following two steps:

1. Visit every variable and class declaration node. Allocate a unique new name for the declaration. Store the relationship between the declaration node and the new name in a map. Replace the declaration with a declaration of the same entity but with the new name.
2. Visit every identifier use node. If the use is of a pre-defined identifier, leave it alone. Otherwise, replace the use with a use of the new name of the entity to which this use is bound.

Step 2 requires us to be able to work out which identifier is bound to a particular identifier use. Recall from Section 3.2 that the PicoJava name analysis specification defines a `decl` attribute of identifier uses that is a reference to the relevant declaration node. Thus, in Step 2 we can search the map created in Step 1 for that declaration node to obtain the new name.

4.3. Implementing the transformation

We implement the transformation using Kiama's version of strategic term rewriting [4, 24]. First, consider the processing of a declaration node in Step 1 of the transformation.

```
val obfuscateDecl =
  rule {
    case d : VarDecl   => d.copy (Name = makeName (d))
    case d : ClassDecl => d.copy (Name = makeName (d))
  }
```

`obfuscateDecl` is a rewrite rule that succeeds only at `VarDecl` or `ClassDecl` nodes. At those nodes, it uses the helper method `makeName` to allocate a new unique name for the declared entity. `makeName` manages a map called `declNames` from declaration nodes to names. (We omit the implementation of `makeName` since it is straightforward.) The `copy` method of the AST node is then used to create a replacement node that is exactly the same as the old one but with the new name.[3] Finally, the rewrite rule succeeds with (returns) the new node which has the effect of replacing the old node with the new node in the transformed tree.

`obfuscateDecl` deals with single declaration nodes and will fail at other nodes. To process all declaration nodes we need a way to traverse the whole tree to find the declarations. We use the generic `topdown` higher-order strategy that is provided by Kiama. `topdown` applies its argument strategy at all nodes in the tree, from the root downwards and in a left-to-right fashion, failing if its argument strategy fails. In the simplest version of this transformation, we apply `obfuscateDecl` at all nodes in the tree and swallow failure. The `attempt` higher-order strategy always succeeds, whether or not its argument succeeds. Thus, we can implement Step 1 of the transformation with the following rewriting strategy.

```
val obfuscateDecls = topdown (attempt (obfuscateDecl))
```

A similar approach suffices to implement Step 2. We divide the basic task into handling pre-defined uses and replacing non-pre-defined uses. We assume that the list `predefinedNames` contains the names that we do not want to replace. `preservePredefinedUse` succeeds at uses of pre-defined names and leaves them unchanged.

```
val preservePredefinedUse =
  rule {
    case u @ Use (name) if predefinedNames contains name =>
      u
  }
```

`obfuscateNormalUse` succeeds at all uses. It uses the `decl` attribute to find the declaration of the entity to which the use refers, and looks it up in the

[3]The expression `d.copy (Name = e)` calls the `copy` method of `d` passing `e` as the value of the named parameter `Name`. In this case the parameter specifies the value of the `Name` field of the new node (see Figure 6).

declNames map created by Step 1.[4] It then succeeds with a new node that is the same as the old node but uses the new name.

```
val obfuscateNormalUse =
  rule {
    case u : Use =>
      u.copy (Name = declNames.getOrElse (u->decl, "$UNDEF$"))
  }
```

We only want to apply obfuscateNormalUse if the check for a pre-defined use does not pass, so we combine the two rules using the left choice operator <+ that only applies its right argument if its left argument fails. As in Step 1, the simplest implementation is to apply the rules to every node in the tree, which results in the obfuscateUses strategy.

```
val obfuscateUses =
  topdown (attempt (preservePredefinedUse <+
                    obfuscateNormalUse))
```

Finally, we construct the complete transformation by combining the implementations of Step 1 and 2 using the sequence operator <* to obtain the obfuscateProgram strategy.

```
val obfuscateProgram = obfuscateDecls <* obfuscateUses
```

4.4. Understanding PicoJava obfuscation

The obfuscation transformation is not a complex one, but its exact operation can be hard to comprehend. For example, how many times and at exactly which nodes are each of the rewrite rules applied? In general, we want to avoid applying rules at nodes where they cannot possible succeed. The brute force topdown (attempt (s)) style of traversal is probably overkill.

To check what actually happens we can use profiles similar to those we saw earlier for attribute evaluations. In this case the events we are interested in are strategy evaluations, which are represented by the event type StratEval in our profiles. The intrinsic dimensions of these events are the strategy that is applied, the subject term to which it is applied, and the result of the application (see the bottom part of Figure 3.)

The top part of Figure 9 shows a profile for a strategy name query produced while obfuscating the program on the left-hand side of Figure 8. (To save space, we have omitted the descendants columns.) We see that the strategies obfuscateProgram, obfuscateUses and obfuscateDecls are applied exactly once as expected. Processing the uses is more expensive than processing the

[4] Scala's getOrElse method for maps takes a key as its first parameter, looks the key up in the map and returns the corresponding entry if one is found. If the key is not present in the map the second parameter is returned.

```
By name for StratEval:

 Total Total  Self  Self  Count Count
    ms     %    ms     %           %
    42  58.4     0   0.2      1   0.1  obfuscateProgram
    42  58.3    12  16.9    150  13.4  result
    41  56.8    10  14.5    150  13.4  all
    22  30.7     0   0.1      1   0.1  obfuscateUses
    19  27.6     0   0.1      1   0.1  obfuscateDecls
    19  26.9    10  15.1    150  13.4  attempt
     7   9.8     5   7.1     75   6.7  <+
     1   2.3     1   2.2     71   6.3  obfuscateNormalUse
     1   1.5     1   1.5     75   6.7  obfuscateDecl
     0   0.6     0   0.6    127  11.3  id
     0   0.4     0   0.4     75   6.7  preservePredefinedUse
```

```
By subject for StratEval and obfuscateNormalUse:

 Total Total  Self  Self  Count Count
    ms     %    ms     %           %
     0   0.9     0   0.9      2   0.2  Use(avar)
     0   0.3     0   0.3      2   0.2  Use(item)
     0   0.3     0   0.3      2   0.2  Use(object)
     0   0.2     0   0.2      2   0.2  Use(bvar)
     0   0.2     0   0.2      1   0.1  Use(ALongClassName)
     0   0.2     0   0.2      1   0.1  Use(NestedClass)
     ...
```

Figure 9: Extracts of profiles produced when the PicoJava obfuscation transformation processes the program on the left-hand side of Figure 8: "event name" for strategy evaluations (top); "event name subject" for the `obfuscateNormalUse` strategy (bottom). (Descendants columns omitted to save space.)

declarations in this run, which makes sense since there are twenty-three uses but only nine declarations.

The tree has seventy-five nodes, so we see that the two full passes result in twice that number of applications of the generic traversal `all` (which is used in the implementation of `topdown`) and `attempt`. `obfuscateDecl` and `preservePredefinedUse` are applied at each node, but `obfuscateNormalUse` is not applied at four nodes. Presumably, the four nodes are the ones for the uses of the `int` identifier. We can check this belief by querying on "event subject". The beginning of a profile for this query is shown in the bottom part of Figure 9. The profile shows that most of the activity occurs at the uses of the non-predefined identifiers, but there is a long tail consisting of almost every

node in the tree. The full profile is missing an entry for `Use(int)` so we can confirm that those four nodes are not processed.

Given this information, we might seek to improve our transformation. For example, the most glaring issue is that the rules are applied at every node in the tree. Can we do better than that? At the very least we could avoid trying to obfuscate normal uses of identifiers if the node is not a use at all. The `preservePredefinedUse` strategy is already acting as a gate-keeper for `obfuscateNormalUse`, but it lets non-`Use` nodes past. We can reduce the number of nodes that `obfuscateNormalUse` has to consider by reversing the sense of the check that `preservePredefinedUse` does. Instead of succeeding on predefined uses, we now succeed only if the subject term is a use that should not be preserved. Non-`Use` nodes will not pass this version of the test. We must also change `obfuscateUses` to use sequencing instead of left choice so that `obfuscateNormalUse` only runs if the new `preservePredefinedUse` succeeds. We now have these improved versions of the strategies.

```
val preservePredefinedUse =
  rule {
    case u @ Use (name) if ! (predefinedNames contains name) =>
      u
  }

val obfuscateUses =
  topdown (attempt (preservePredefinedUse <*
                    obfuscateNormalUse))
```

With these changes the profile shows that `obfuscateNormalUse` only runs at ten nodes, exactly those that are shown in the bottom part of Figure 9 and excluding the long tail.

These kinds of optimisations can make a big difference when we are working with complex transformations. Profiles of the kind shown here can focus our attention on the strategies that are doing the most work and help us confirm that our changes have made a positive difference.

4.5. Debugging with domain-specific profiles

The `topdown-attempt` style of traversal used in the obfuscation rewrites is a common pattern, but it can be difficult to know when to use it. We now show an example where a missing `attempt` is not obvious until we profile. This example illustrates a common error in rewriting programs.

PicoJava allows classes to be nested inside one another. We would like to define a code transformation that removes nested classes, making their contents part of the enclosing class. We identify the class to remove by giving its name.

We define a rewrite rule that looks for class declarations in a list of block statements. We only need to identify class declarations at the start of such a list because recursive application of the strategy will be used to ensure that each

sub-list is checked. In the case when we find such a declaration, we replace it with its body by pre-pending the body to the rest of the current declarations.[5]

```
def removeClass (name: String) =
  rule {
    case (ClassDecl(‘name‘, _, Block (bs)) :: rest) =>
      bs ++ rest
  }
```

All that is left now is to ensure that this rewrite rule is applied over the entire program. A common choice is to use the `topdown` generic traversal as used in the obfuscation transformation. We assume in this definition that `name` contains the name we wish to search for.

```
val removeClasses = topdown (removeClass (name))
```

However, this code fails, not making any changes at all when applied to the program in Figure 5 if we ask it to remove classes named `AA`. This problem is quickly confirmed by the extract from a profile shown in the top part Figure 10 which shows that `removeClass` has only been applied once.

```
    0   0.3     0   0.2     0   0.2     1   0.4  topdown
    0   0.2     0   0.2     0   0.0     1   0.4  removeClass

    8  22.9     2   7.5     5  15.4    58  11.1  topdown
    0   1.7     0   1.7     0   0.0    58  11.1  removeClass
    0   1.4     0   1.4     0   0.0    57  10.9  id
```

Figure 10: Two extracts from profiles of the `removeClasses` strategy when applied to the program in Figure 5. The top extract shows the erroneous situation when the recursive applications are not being performed. The bottom extract shows the correct behaviour when the rewrite is performed throughout the tree.

We expected that the use of the top-down strategy would have applied it to all nodes, but it was applied just once. We have found an instance of a common term rewriting error, *unexpected failure*, which has been identified by Lämmel *et al.* as a common cause of bugs in rewriting programs [25]. `removeClass` is a strategy that fails when not applicable and the `topdown` strategy stops at failure. We require a version of `removeClass` which does nothing when not applicable.

We can create a version of `removeClass` that leaves its input alone when not applicable by wrapping it in the strategy `attempt` as we did in the obfuscation example.

[5]Scala allows pattern matching on a specific named value by enclosing the name in back-quotes as in `‘name‘`. The `::` operator is the list constructor taking a list element on the left and a list on the right. List concatenation is performed by the `++` operator.

```
val removeClasses = topdown (attempt (removeClass (name)))
```

The bottom part of Figure 10 confirms that the behaviour is more reasonable. `removeClass` is now applied fifty-eight times, once for each node in the abstract syntax tree. We can also learn from the rest of the profile, such as the number of times that the identity strategy `id` was applied by `attempt`. In this case the difference between the number of times that `removeClass` was applied and the number of applications of `id` was one, so we can see that `removeClass` succeeded exactly once. This is the expected result for this example since we were removing a single class.

5. Discussion and Related Work

The vast majority of previous work in the profiling area has concentrated on execution profiles for programs, inspired by systems such as the seminal `gprof` tool [1]. A gprof-style profile shows the functions that were called by the program and aggregates the execution time of each function according to time used by that function itself and time used by the functions it called. The profile also shows the actual pattern of control flow between functions so that the behaviour of a function can be analysed in terms of how much time is spent using other functions.

Our approach is a generalisation of the gprof approach. Instead of restricting our attention to constructs such as functions that appear in the source code of a program, we allow attention to be brought to bear on any domain-specific operation. Instead of summarising control flow in terms of calls between functions, we allow arbitrary hierarchical dependencies between domain-specific operations. Instead of listing the functions that were called, we allow the profile to report on execution events in terms of domain-specific dimensions.

While the approach is similar, it is important to appreciate the major differences. A gprof-style profile reports on execution in terms of the source code form of the program. Our approach allows DSL developers to build profiling support that reports using any events and dimensions that are appropriate for their domain. DSL users can even extend the profiler by adding new derived dimensions. There need not be a direct relationship between the code of a DSL program and the events and dimensions. For example, in our attribute grammar profiles there is nothing in a DSL program that talks about attribute caches, yet it is part of the dynamic attribute grammar domain. Making the caching behaviour part of the attribute evaluation events is natural since it is part of the attribute grammar domain.

Furthermore, there need not be any simple relationship between the functions (or methods) called by the DSL implementation and the events that it generates. This observation precludes a simple alternative approach where the output of a gprof-style profiler is mapped into domain-specific operations or a compiled program is instrumented at run-time. Our instrumentation approach is easy to use and extremely flexible since events can be generated from anywhere in the DSL implementation. (Our presentation has focused on DSLs that

are embedded in a general-purpose language but a DSL implementation built around translation or interpretation could use a very similar approach.) The price we pay is that the DSL implementation must be modified, but since the developer of the profiling facility is very likely to be the DSL developer, we do not think this limitation is a major one.

5.1. Value profiling

Value profiling has been investigated for applications in low-level compiler optimisation [32]. The context of value profiling is quite different to our work, since the profiles are obtained by instrumenting load instructions and memory locations. The basic result is similar, though, since in both cases we show that examining the values that are actually used in a program can reveal information that aids in understanding how that program runs.

5.2. Abstraction in profiling systems

Instead of profiling at the function level, our approach raises the level of abstraction. Abstraction increases the generality of the profiling system and significantly reduces the size of the collected data compared to instruction and function-level profilers.

Sansom and Peyton Jones describe a profiler for higher-order functional languages including Haskell [33]. The execution of higher-order programs, particularly lazy ones, is not obvious since the compilation process is non-trivial and execution order often does not correspond clearly to the source code. They allow developers to add "cost centres" that aggregate data in a program-specific manner. Thus, the source-level profile data can be lifted to a higher level. In our case, the data is always at a higher level. Their cost centres are each associated with source code fragments rather than separated into `Start` and `Finish` events as in our approach. Thus, the identification of an abstracted piece of program execution is more flexible in our approach because the two events do not have to be associated with the same source code.

Nguyen *et al.* describe a domain-specific language for automating the regulation of profile data collection, processing and feedback [34]. The language allows some abstraction away from the details of the profile exploration process. However, it is different from our work in that it operates at a low level and is intended for analysing performance of programs and system kernels in a similar fashion to `gprof`.

Rajagopalan *et al.* consider profiling for event-based programs such as graphical user interfaces [35]. Events in their work are intrinsic to the functioning of the system, whereas in our work they are solely part of the profiling system. They look for patterns in the events which allows them to abstract away from the execution somewhat, but they do not consider a general abstraction mechanism.

The DTrace tracing framework provides powerful facilities for inserting probes to collect data about programs as they run [36]. Systems such as the Linux Trace Toolkit [37] and the Java Virtual Machine Tool Interface can also be used to

collect a large amount of trace data about program execution. These mechanisms enable profiling data to be collected, but they operate at a much lower level than the DSL level that we operate at in our approach.

For example, to build an attribute evaluation profiler for Kiama programs using DTrace we would have to insert probes at the Java Virtual Machine (JVM) level and write code in DTrace's D language to assemble the data obtained from these probes into DSL-level information such as the hierarchy of attribute evaluations, attribute names, etc. Just the task of inserting the probes appropriately would require very specific knowledge of how the Kiama Scala code has been translated down to JVM byte code. Thus, the task of adding profile support to a DSL implementation would be much harder than in our approach, which does not require the developer of the DSL to do anything more than insert a small number of method calls in their DSL implementation. Moreover, DTrace and similar tools require a lot of infrastructure support from the system on which they run, so they are a much more heavyweight profiling solution than our approach. Our dsprofile library is small and easy to understand and could easily be implemented on non-JVM platforms. This lightweight nature suggests that the approach is more appropriate for situations where DSLs are being developed and deployed rapidly.

Bergel *et al.* describe a model-based domain-specific profiling approach [38]. Instrumentation code augments domain model code via an existing event mechanism or by using the host language's reflection framework. Custom profilers use the information gathered to display profiles and visualisations that relate directly to domain data and operations. The reliance on meta-programming features of the framework distinguishes their approach from ours. We pay the price of having to instrument the DSL implementation by hand, but as a result we make no demands on the underlying runtime. The dsprofile library is independent of any particular domain and the reports have a generic format, whereas their profilers are deliberately tailored to particular model code and particular kinds of observations that they want to make.

5.3. Profiling attribute grammars

Saraiva and colleagues have investigated the efficiency of attribute grammar evaluation approaches, notably as part of work to improve the efficiency of evaluators that are constructed as circular programs [39]. Their experiments focused on course-grained measures such as heap usage, rather than the kind of fine-grained analysis considered in this paper. In earlier work, Saraiva compared the performance of functional attribute evaluators [40]. He examined properties such as hash table size, cache misses and the number of equality tests performed between terms for both full and incremental evaluation of attributes. Some of these measures have analogues in our approach. The implementation appears to be custom to the particular experiment rather than a general facility as in our library.

Söderberg and Hedin show how attribute profiles can be used to analyse caching behaviour in JastAdd [41]. They calculate an *attribute instance graph* that is an attribute dependence graph with evaluation counts on the edges.

Edges labelled with counts greater than one point to attributes for which caching might be advantageous. Their attribute dependence graph is similar to our collection of profile records and dependency relationships, except that our records are independent of the attribute evaluation domain. Our use of arbitrary dimensions to extend the power of profiles goes beyond the aim of Söderberg and Hedin's work which was to look solely at caching issues.

In earlier work, the first author developed the Noosa execution monitoring system for the Eli system, including the attribute grammar component [42]. Noosa is a debugging system, not a profiler, but it also uses an event-based approach to record information about the execution of a program. Noosa doesn't group events in the same way as the Kiama profiler, since it uses events primarily to specify domain-specific breakpoints. The focus is on controlling the execution as it happens rather than on summarising it after it is done. Noosa can be used to examine the values of attributes of interest with reference to the abstract syntax tree, but it cannot be used to summarise the execution along other dimensions.

5.4. Profiling rewriting systems

Some rewriting systems have very sophisticated profiling systems, but none are generic enough to encompass other facets of computation (like attribute grammars) as ours does. Stratego has the ability to print tracing information as a program is executed [43], but this information can't be interrogated in different ways and requires many program annotations. Maude has an extensive profiling system built specifically to capture rewriting information [44] from which we plan to take inspiration as we extend the profiling capabilities of Kiama. However, profiling in Maude is restricted to term rewriting and there is no way to profile attributes or other hierarchical computation. Other rewriting systems rely on the capabilities of the underlying implementation language and thereby expose some implementation details to the programmer. For example, the Tom system [45] is a moderate extension of Java so profilers for that language can be used to examine the behaviour of Tom programs.

Van den Brand *et al* [46] show how to create a debugger that will work not just for rewriting, but for systems built with rewriting. The TIDE system they created allows you to create breakpoints and to see the source code during debugging. Our work focusses on profiling as opposed to debugging, including execution time; plus our work allows post-facto analysis of the collected data.

Like our profiler, all of the above analyses are dynamic. Lämmel *et al.* [25, 26] have catalogued many kinds of errors that occur in programs using strategic rewriting and have developed static analyses to discover errors via the type system and to guide repair efforts. Our work is complementary to static analysis. Ultimately, we imagine a system where static analysis finds the errors it can and the remaining are left to dynamic analysis tools. As we have shown in Section 4 our profiles can already be used to help diagnose some rewriting problems of the kind identified by Lämmel *et al.*

6. Conclusion and Future Work

We have described a new general approach to domain-specific profiling and its application to profiling software language implementations that are based on dynamically-scheduled attribute grammar evaluators and strategic rewriting systems. The approach is easy to implement and its execution overhead is low enough for interactive use. We have described applications of the profiler to program understanding, testing and debugging.

One direction for future work is to deploy the profiler with other attribute grammar systems. The approach used with Kiama should easily transfer to other systems based on dynamic scheduling, but minor modifications should allow it to be used with other evaluation approaches. For example, a statically-scheduled tree walking attribute evaluator could be instrumented automatically by the scheduler. There might be scope to add new dimensions, such as one that captures information about node visits across different attributes.

On the rewriting side, we think that is worthwhile to explore the behaviour of rewrite strategies at both higher and lower levels we have considered in this paper. At a higher level it should be possible to report on general traversal patterns by combining information from multiple events. At a lower level it will be useful to consider strategy failure in more detail. Strategies can fail when the subject term does not match any of the applicable patterns. Higher-order strategies can fail due to the operation of one of their argument strategies. Failure is a primary concern in strategic rewriting because it drives choices in the execution path, particularly during generic traversals. The profiling support currently implemented in Kiama captures failure of a strategy application via its result intrinsic dimension but we have not explored its use for non-local failure analysis.

The execution model we use is not tied to attribute grammars or rewriting systems, so we expect it to be useful for profiling code that is based around other abstractions. For example, we could imagine generating events as a parser executes so that dynamic behaviour such as back-tracking could be examined.

[1] S. L. Graham, P. B. Kessler, M. K. Mckusick, gprof: A call graph execution profiler, in: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, SIGPLAN '82, ACM, New York, NY, USA, 120–126, 1982.

[2] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-Specific Languages, Computing Surveys 37 (4) (2005) 316–344.

[3] M. Odersky, L. Spoon, B. Venners, Programming in Scala, Artima Press, 2nd edn., 2010.

[4] A. M. Sloane, Lightweight Language Processing in Kiama, in: Generative and Transformational Techniques in Software Engineering III, vol. 6491 of *Lecture Notes in Computer Science*, Springer, 408–425, 2011.

[5] J. Paakki, Attribute grammar paradigms—a high-level methodology in language implementation, Computing Surveys 27 (2) (1995) 196–255.

[6] B. Bouchou, M. Halfeld Ferrari, M. Lima, Attribute Grammar for XML Integrity Constraint Validation, in: Database and Expert Systems Applications, vol. 6860 of *Lecture Notes in Computer Science*, Springer, 94–109, 2011.

[7] D. Davidson, R. Smith, N. Doyle, S. Jha, Protocol Normalization Using Attribute Grammars, in: Computer Security – ESORICS 2009, vol. 5789 of *Lecture Notes in Computer Science*, Springer, 216–231, 2009.

[8] T. Ekman, G. Hedin, The JastAdd extensible Java compiler, in: Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, ACM, New York, NY, USA, ISBN 978-1-59593-786-5, 1–18, 2007.

[9] F. Han, S.-C. Zhu, Bottom-Up/Top-Down Image Parsing with Attribute Grammar, IEEE Transactions on Pattern Analysis and Machine Intelligence 31 (1) (2009) 59 –73.

[10] M. Karim, C. Ryan, A New Approach to Solving 0-1 Multiconstraint Knapsack Problems Using Attribute Grammar with Lookahead, in: Genetic Programming, vol. 6621 of *Lecture Notes in Computer Science*, Springer, 250–261, 2011.

[11] A. M. Sloane, L. C. L. Kats, E. Visser, A pure embedding of attribute grammars, Science of Computer Programming 78 (2013) 1752–1769.

[12] E. Magnusson, G. Hedin, Circular reference attributed grammars–their evaluation and applications, Science of Computer Programming 68 (1) (2007) 21–37.

[13] H. H. Vogt, S. D. Swierstra, M. F. Kuiper, Higher order attribute grammars, in: PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, ACM Press, 131–145, 1989.

[14] U. Kastens, Ordered Attribute Grammars, Acta Informatica 13 (1980) 229–256.

[15] G. Hedin, E. Magnusson, JastAdd: an aspect-oriented compiler construction system, Science of Computer Programming 47 (1) (2003) 37–58.

[16] M. Jourdan, An optimal-time recursive evaluator for attribute grammars, in: Proceedings of the International Symposium on Programming, Springer, 167–178, 1984.

[17] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: An extensible attribute grammar system, Science of Computer Programming 75 (1+2) (2010) 39–54.

[18] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[19] E. Visser, Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9, in: C. Lengauer, et al. (Eds.), Domain-Specific Program Generation, vol. 3016 of *Lecture Notes in Computer Science*, Springer-Verlag, 216–238, 2004.

[20] E. Visser, Domain-Specific Language Engineering, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Universidade do Minho, Braga, Portugal, 265–318, 2007.

[21] J. Meseguer, Twenty years of rewriting logic, The Journal of Logic and Algebraic Programming 81 (7–8) (2012) 721 – 781.

[22] L. C. L. Kats, E. Visser, The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs, in: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 444–463, 2010.

[23] R. E. Shostak, Applying term rewriting to speech recognition of numbers, in: Formal Methods and Software Engineering, Springer, 4–4, 2012.

[24] S. Premaratne, A. M. Sloane, L. G. C. Hamey, An Evaluation of a Pure Embedded Domain-Specific Language for Strategic Term Rewriting, chap. 4, Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 81–108, 2013.

[25] R. Lämmel, S. Thompson, M. Kaiser, Programming Errors in Traversal Programs Over Structured Data, Electron. Notes Theor. Comput. Sci. 238 (5) (2009) 135–153.

[26] A. Mametjanov, V. Winter, R. Lämmel, More precise typing of rewrite strategies, in: Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA '11, ACM, 3:1–3:8, 2011.

[27] A. M. Sloane, Profile-based Abstraction and Analysis of Attribute Grammar Evaluation, in: Proceedings of the 2012 International Conference on Software Language Engineering, vol. 7745 of *Lecture Notes in Computer Science*, Springer, 24–43, 2013.

[28] E. Burmako, Scala Macros: Let Our Powers Combine!, in: Proceedings of the 4th Annual Scala Workshop, ACM, 2013.

[29] N. Wirth, Compiler Construction, Addison-Wesley, revised Nov. 2005, 1996.

[30] G. Hedin, Reference Attributed Grammars, Informatica 24 (3) (2000) 301–317.

[31] U. Kastens, W. M. Waite, Modularity and Reusability in Attribute Grammars, Acta Informatica 31 (1994) 601–627.

[32] B. Calder, P. Feller, A. Eustace, Value profiling and optimization, Journal of Instruction Level Parallelism 1 (1-6) (1999) 4J.

[33] P. M. Sansom, S. L. Peyton Jones, Formally based profiling for higher-order functional languages, ACM Trans. Program. Lang. Syst. 19 (2) (1997) 334–385.

[34] P. Nguyen, K. Falkner, H. Detmold, D. S. Munro, A domain specific language for execution profiling & regulation, in: Proceedings of the Australasian Conference on Computer Science, Australian Computer Society, Inc., 123–132, 2009.

[35] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, R. D. Schlichting, Profile-directed optimization of event-based programs, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI '02, ACM, New York, NY, USA, 106–116, 2002.

[36] J. Mauro, B. Gregg, C. Mynhier, DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD, Prentice Hall Professional, 2011.

[37] K. Yaghmour, M. R. Dagenais, The linux trace toolkit, Linux Journal, May, 2000 106.

[38] A. Bergel, O. Nierstrasz, L. Renggli, J. Ressia, Domain-Specific Profiling, in: J. Bishop, A. Vallecillo (Eds.), Objects, Models, Components, Patterns, vol. 6705 of *Lecture Notes in Computer Science*, Springer, 68–82, 2011.

[39] J. a. P. Fernandes, J. a. Saraiva, D. Seidel, J. Voigtländer, Strictification of circular programs, in: Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation, PEPM '11, ACM, New York, 131–140, 2011.

[40] J. Saraiva, Purely Functional Implementation of Attribute Grammars, Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands, 1999.

[41] E. Söderberg, G. Hedin, Automated Selective Caching for Reference Attribute Grammars, in: Proceedings of the 3rd International Conference on Software Language Engineering, 2–21, 2010.

[42] A. M. Sloane, Debugging Eli-generated compilers with Noosa, in: Proceedings of the 8th International Conference on Compiler Construction, vol. 1575 of *Lecture Notes in Computer Science*, Springer, 17–31, 1999.

[43] M. Bravenboer, K. T. Kallebarg, E. Visser, Debugging Techniques for Stratego/XT, chap. 30, Stratego/XT Manual, 2009.

[44] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, Debugging and Troubleshooting, chap. 14, Maude Manual, 2010.

[45] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, Tom: Piggybacking rewriting on Java, in: Term Rewriting and Applications, Springer, 36–47, 2007.

[46] M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, J. J. Vinju, TIDE: A Generic Debugging Framework (Tool Demonstration), Electronic Notes in Theoretical Computer Science 141 (4) (2005) 161–165.