A formal definition of crosscuts[*]

Rémi Douence, Olivier Motelet, Mario Südholt

École des Mines de Nantes

4 rue Alfred Kastler, 44307 Nantes cedex 3, France

`www.emn.fr/{douence,motelet,sudholt}`

Abstract

Crosscutting, i.e. relating different program points is one of the key notions of Aspect-Oriented Programming. In this report, we consider a general and operational model for crosscutting based on execution monitors.

A domain-specific language for the definition of crosscuts constitutes the core of the report. The semantics of this language is formally defined by means of parser operators matching event patterns in execution traces. We define an operational semantics of the matching process by means of rules relating the operators and illustrate how to use these rules to formally prove optimization properties. The use of the language is exemplified by several sophisticated crosscut definitions. We present a prototype implementation in JAVA which has been systematically derived from the language definition.

Technical report no.: 01/3/INFO

[*]This work has been partially funded by the EU project "EasyComp" (`www.easycomp.org`), no. IST-1999014191.

Contents

1 Introduction **3**

2 AOP from a monitoring perspective **3**
2.1 Overview of the framework 4

3 A formal definition of crosscuts **6**
3.1 A formal framework for AOP 6
3.2 A domain-specific language for crosscut definition 7
3.3 Crosscut definitions 10

4 A JAVA prototype **13**
4.1 Architecture 13
4.2 Static instrumentation 14
4.2.1 Event representation 14
4.2.2 Program instrumentation using COMPOST 14
4.3 Dynamic monitoring 18
4.3.1 Data representation 18
4.3.2 Matching patterns 19
4.3.3 Performing actions 19

5 Related Work **22**

6 Conclusion **24**

A HASKELL implementation of a DSL for crosscutting **27**

B Proof of an optimization property involving `cflow` **30**

1 Introduction

Aspect-Oriented Programming (AOP) [9] is an emerging programming paradigm providing explicit support for separation of concerns. Once different concerns have been identified, two main questions arise. Where non-functional concerns must be inserted in the base program (aka. the functional aspect)? How the non-functional aspects interact with the functional one and with one another? In this report, we focus on the first question. More specifically, we are interested in an operational definition of crosscuts, i.e., relating different points in the execution of a sequential program. We argue for a formal definition of crosscuts for two reasons. First, well-defined crosscutting is a mandatory requirement for the understanding of AOP tools. Second, a formal definition enables aspect composition and compilation to be treated systematically.

In this report, we argue for execution monitors as a general and operational model for AOP and, in particular, crosscutting. Indeed, the "points of interest" of the execution can be abstracted as events and we present a specialized language which enables quite general relations between events to be defined. Moreover, this language provides an operational semantics for crosscut detection.

The report is structured as followed: Section 4.3 introduces execution monitoring as a model for AOP. Section 3 presents a rigorous treatment of the framework, formally defines our crosscutting language and presents several examples of crosscut definitions. Section 4 describes a prototype implementation of our framework in JAVA. Related work is discussed in Section 5. Section 6 concludes and lists future work. Two appendixes give an implementation of our framework in HASKELL and a formal proof of an optimization property, respectively.

2 AOP from a monitoring perspective

Crosscutting is one of the key notions of AOP: a crosscut relates different program points or execution points, such that functionality can be parameterized with information from some of these points and inserted at others. The relations between program points can be quite diverse. For instance, an aspect for method logging requires only individual calls to be identified in order to construct and write the corresponding log message. On the other hand, multi-step security protocols require more complex relationships. During execution of a securized application, such protocols relate points where certificates are generated, points where authentication based on certificates is done and points where resource accesses require that specific rights have been granted after authentication.
In this report, we use *billing* in (what we call) "loosely-coupled client-server systems" as a running example. In such client-server systems, communication is initiated and mediated by a controller. Figure 1 shows a controller that is intended to manage a communication between customers and services which should not or cannot communicate directly. A typical scenario of such a system architecture is the following: first select a customer request, second prepare the service call by performing a potentially complex computation

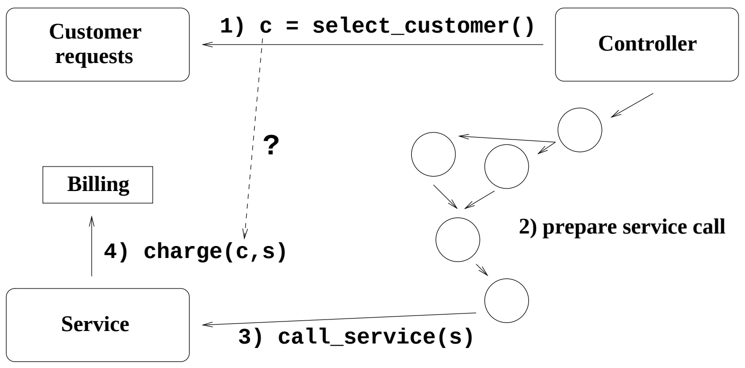

Figure 1: Billing in a loosely-coupled client-server system

(e.g. establishing communication statistics or performing configuration tasks) and third call a service. In such an application, the last selected customer is not available when a service is called. However, this information is mandatory in order to upgrade the application such that the service (not the controller!) can bill the customer.

A conventional solution to this problem is to propagate the customer identity by threading this piece of information through the complex computation code (e.g. add an extra argument to all functions implementing this code). Obviously, such a modification can be rather tedious depending on the size of the code. An AOP-like solution is to define a billing crosscut that relates a customer request selection to the corresponding service called. Such systems thus provide an interesting and complex example for crosscutting with realistic applications.

2.1 Overview of the framework

The examples we just discussed motivate several essential characteristics any framework for AOP should have: first, a mechanism which allows programs (i.e. program code) or program executions to be *abstracted* to "points of interests," i.e., points where information is needed from or points where new behavior is to be inserted. Second, a part of the framework that must be able to *relate* such points of interests. Third, this part must also be able to gather the information provided at the points of interest and insert the new behavior, i.e., to *perform* actions.

Monitor-based AOP. We advocate the use of execution monitors as a framework for AOP. A monitor has a global view on the program execution. In this context, the framework characteristics are materialized as follows:

1. The points of interest of a program execution are defined in terms of *events* emitted during program execution.

2. Points of interest that relate to one another are denoted by *patterns of events* to be *matched.*

3. Once a pattern has been matched, the program execution is suspended and an *action* may be executed.

We claim that execution monitors are highly appropriate as a framework for AOP. They provide a natural abstraction in terms of events, enable the explicit definition of complex crosscuts by means of event patterns and accommodate very general actions. In this report, we are mainly interested in substantiating this claim by giving a formal definition of crosscutting as event patterns and the process of pattern matching.

Crosscut definitions. In order to get a first intuitive understanding of the kind of crosscuts we are interested in, let us consider the billing example introduced above. The essence of this example is to relate customer selections to service calls despite the loose coupling between clients and servers. Let us therefore assume that two corresponding events $select\_customer()$ and $call\_service(s)$ are generated during program execution. Once these events are generated, we may want to match the following event patterns:

- A pair consisting of a customer selection and the next service call.
- A (possibly infinite) sequence of such pairs — e.g., to cope with iteration in the controller.
- Sequences of overlapping pairs of selection and call — e.g., to cope with buffering in the controller: a number of customer selections is followed by the corresponding service calls. By means of overlapping pairs, we can disambiguate the sequence of events

  $$\begin{array}{l} c_1 = select\_customer() \ldots c_2 = select\_customer() \\ \ldots call\_service(s_1) \ldots call\_service(s_2) \end{array}$$

  as relating c_1 to s_1 as well as c_2 to s_2 (but not, e.g., c_1 to s_2).
- Nested pairs of customer selections and service calls — e.g., to cope with recursion in the controller: a customer selection may require that another customer-service pair is executed first.

Aspect definitions. An aspect can then be defined as a rule $pattern \Rightarrow action$ and several aspects can be defined simply as concurrent rules. A billing aspect, for instance, could be expressed by the rule

$$c = select\_customer();\ call\_service(s)\ \Rightarrow\ charge(c, s)$$

which can be interpreted as follows: monitor the application execution in order to detect the next call to the $select\_customer()$ function, pause the execution, store the value of the variable c, resume execution, monitor the execution in order to detect the next call to the $call\_service()$ function, pause the execution, store the value of the variable s, call the function $charge()$ and resume execution. Hence, the pattern on the left-hand side of the rule relates every customer selection to the next service called during program execution.

3 A formal definition of crosscuts

In this section, we propose a formal definition of crosscuts: Section 3.1 shows how the main notions of our framework can be formalized. Section 3.2 presents our domain-specific language for crosscutting and the properties of its operators. Finally, Section 3.3 applies this language to the billing example and sketches how formal properties of the pattern operators can be used.

In the following, we present a formalization using the functional language HASKELL [6]. We introduce its syntactic elements on the fly as needed. The full implementation defining our framework can be found in Appendix A.

3.1 A formal framework for AOP

In our framework, an event represents a point in the program execution. The following type synonym models events by a pair consisting of a name tag and a time stamp.[1]

```
type Event = (String, Int)
```

The complete trace of the execution is defined as a list of events.

```
type Trace = [Event]
```

A program is a sequential computation that produces a trace, one event at a time.

```
data Program = Over | Cont (Tick -> (Event, Program))
```

This is modeled by the following algebraic data type: either the program terminates (represented by the constructor `Over`) or it is a continuation (`Cont`) that at each step (represented by `Tick`) returns the next event and the remainder of the program.

A crosscut is defined as a list of events representing different points in the program execution which are to be related.

```
type Crosscut = [Event]
```

A monitor is a program that performs event pattern matching, one event at a time. It can be defined as a function mapping an event to a pair consisting of a list of crosscuts detected at this event and the monitor continuation.[2]

```
data Monitor = M (Event -> ([Crosscut], Monitor))
```

[1]A more realistic event definition — extending the one given in this section — is discussed along with the presentation of our JAVA prototype in Section 4.

[2]In HASKELL, type synonyms cannot be recursive and the constructor `M` is thus needed for technical reasons.

The program and the monitor implement a producer-consumer pair. In order to compose them, we define a function `run`[3] that returns the list of all crosscuts detected during program execution:

```
run :: Program -> Monitor -> [Crosscut]
run Over           _           = []
run (Cont program) (M monitor) =
      let (event,program')     = program Tick
          (crosscuts,monitor') = monitor event
       in crosscuts ++ (run program' monitor')
```

When the program execution is over, an empty list of crosscuts is returned. Otherwise, the program generates an event which is passed to the monitor and the list of detected crosscuts at this point is concatenated (++) with the crosscuts to come (recursive call to `run`).

Note that we focus on the definition and detection of crosscuts in this report. So, we do not model actions which are to be interleaved with the program execution each time a crosscut is detected. However, in order to define the framework completely, aspect actions could be modeled as functions mapping a crosscut and the current program state to another program state:

```
type Action = Crosscut -> Program -> Program
```

3.2 A domain-specific language for crosscut definition

Rephrasing the definition of the type `Monitor` above, we can say that the monitor is a function of past events. This function can be as complex as needed in order to detect sophisticated crosscuts. However, in order to support a concise definition of crosscuts, we argue for a domain-specific language (DSL) to define event patterns.
In Figure 2, we present such a DSL for crosscut definitions that is based on the notion of event patterns. Pattern constructors are classified in four categories: sequential patterns, filtering patterns, abortable patterns and parallel patterns.

Informally, our constructions can be interpreted as follows: `Return c` returns `c` as a result; `Bind f` applies the function `f` to the previous result (and thus binds it to a variable); `p1 ‘Seq‘ p2`[4] matches first `p1` and then `p2`; `Filter pred p1` filters out the next event that satisfies `pred` while matching `p1`; `Abort` aborts the current sequential pattern; `p1 ‘Par‘ p2` matches `p1` and `p2` simultaneously; `First p` returns the result of the first sequential pattern of (a `Par`-expression) `p` that matches.

In order to familiarize the reader with this language, we present a few basic pattern/crosscut definitions. (More sophisticated patterns are presented in the following section.)

[3]In HASKELL, a function definition is preceeded by its type (introduced by `::`).

[4]In HASKELL, `‘C‘` denotes infix application of `C`.

```
data Pattern =
      -- sequential patterns
      Return Crosscut
    | Bind (Crosscut -> Pattern)
    | Pattern `Seq` Pattern
      -- filtering patterns
    | Filter (Event -> Bool) Pattern
      -- abortable patterns
    | Abort
      -- parallel patterns
    | Pattern `Par` Pattern
    | First Pattern
```

Figure 2: DSL for crosscut definition

- A pattern defining a crosscut consisting only of the next event.

  ```
  nextEvent :: Pattern
  nextEvent = Bind (\[e] -> Return [e])
  ```

  This pattern binds the next event to the variable `e`[5] and returns it as a crosscut.

- A pattern that matches the next event satisfying a predicate `pred`.

  ```
  nextP :: (Event -> Bool) -> Pattern
  nextP pred = Bind (\[e]-> if (pred e) then Return [e]
                                          else nextP pred)
  ```

  The pattern `nextP` binds the next event and returns it as a crosscut if the predicate is satisfied, otherwise the pattern is called recursively.

- A pattern that matches the next two events and returns the corresponding crosscut.

  ```
  nextThenNext :: Pattern
  nextThenNext = nextEvent `Seq` Bind (\[e1] ->
                 nextEvent `Seq` Bind (\[e2] ->
                 Return [e1,e2]))
  ```

- A pattern that matches the second event satisfying a predicate by filtering out the first one.

[5]In HASKELL, the syntax `\arg->exp` denotes the (anonymous) function $\lambda arg.exp$.

```
second :: (Event -> Bool) -> Pattern
second pred = Filter pred (nextP pred)
```

- A pattern that matches the next event and returns it provided the event does not satisfy a predicate, but does not return a crosscut otherwise.

  ```
  no :: (Event -> Bool) -> Pattern
  no pred = nextEvent ‘Seq‘ Bind (\[e] ->
            if (pred e) then Abort
                        else return [e])
  ```

  Note that this pattern is different from `nextP (not . pred)`[6] that returns the next event that does not satisfies `pred`. Indeed, if the next event satisfies `pred`, `no pred` aborts, while `nextP (no . pred)` skips the event.

- A pattern that matches the next event satisfying `pred1` *and* matches the next event satisfying `pred2`. Note that in this case, two different crosscuts are returned.

  ```
  and :: (Event -> Bool) -> (Event -> Bool) -> Pattern
  and pred1 pred2 = nextP pred1 ‘Par‘ nextP pred2
  ```

- A pattern that matches *either* the next event satisfying `pred1` *or* the next event satisfying `pred2` whichever matches first. In this case only one crosscut is returned.

  ```
  or :: (Event -> Bool) -> (Event -> Bool) -> Pattern
  or pred1 pred2 = First (and pred1 pred2)
  ```

Formal semantics. Figure 3 lists the equivalence rules which the pattern operators obey. The sequential composition of pattern is associative (R1). A sequence of `Return` and `Bind` is equivalent to applying the function to the returned result (R2). `Abort` terminates a sequence (R3) but does not interfere with the parallel composition (R8). The parallel composition of patterns is associative (R4) and commutative (R5) and is distributive with respect to sequential composition (R6, 7). Rules R9-16 deal with `First` and `Filter`. Rules R9 and 13 state how results are propagated under these two constructions. The remaining rules define the semantics of `First` (that returns the result of the first sequential pattern that matches within a `Par`-expression) and `Filter` (that filters out the next event that satisfies a predicate).
The equivalence rules shown in Figure 3 can be used as the essential building block of an interpreter for pattern terms. In the HASKELL implementation shown in Appendix A, the function `monitor :: Pattern -> Monitor` interprets terms of type `Pattern` by reducing a term of the form `Return [e] ‘Seq‘ pattern` where `e` is the current event. The function `monitor` implements a term-rewriting system obtained from the equivalence rules (see the definition of the function `step` in Appendix A).

[6]In HASKELL, ’ .’ denotes function composition.

```
R1   p1 ‘Seq‘ (p2 ‘Seq‘ p3) = (p1 ‘Seq‘ p2) ‘Seq‘ p3
R2   Return es ‘Seq‘ Bind f = f es
R3   Abort ‘Seq‘ p2 = Abort
R4   p1 ‘Par‘ (p2 ‘Par‘ p3) = (p1 ‘Par‘ p2) ‘Par‘ p3
R5   p1 ‘Par‘ p2 = p2 ‘Par‘ p1
R6   p1 ‘Seq‘ (p2 ‘Par‘ p3) = (p1 ‘Seq‘ p2) ‘Par‘ (p1 ‘Seq‘ p3)
R7   (p1 ‘Par‘ p2) ‘Seq‘ p3 = (p1 ‘Seq‘ p3) ‘Par‘ (p2 ‘Seq‘ p3)
R8   Abort ‘Par‘ p2 = p2
R9   p1 ‘Seq‘ First p2 = First (p1 ‘Seq‘ p2)
R10  First (Abort) = Abort
R11  First (Return e) = Return e
R12  First (Return e ‘Par‘ p2) = Return e
R13  Return e ‘Seq‘ Filter pred p2
         = p2                                -- if (pred e)
         = Filter pred (Return e ‘Seq‘ p2)   -- otherwise
R14  Filter pred (Abort) = Abort
R15  Filter pred (Return e) = Return e
R16  Filter pred (Return e ‘Par‘ p2) = Return e ‘Par‘ Filter pred p2
```

Figure 3: Equivalence rules for the event pattern operators

3.3 Crosscut definitions

In this section, we formally define the *billing* crosscuts introduced in Section 4.3 and exemplify how formal properties of the pattern operators can be used in this application context. We assume in the remainder of this section that customer selections and service calls are modeled by events tagged "c" and "s", respectively. A first simple crosscut for billing consists in matching a customer selection followed by the next service selection.

```
billing :: Pattern
billing =
    next "c" ‘Seq‘ Bind (\[e1] ->
    next "s" ‘Seq‘ Bind (\[e2] ->
    Return [e1,e2]))
```

where `next n` matches (by calling `nextP`) the next event named `n`. Matching the pattern `billing` in the trace "`cscs`" yields the single crosscut `[("c",1),("s",2)]`.

Of course, there can be multiple occurrences of this pattern in the execution trace. In order to detect all the crosscuts, it is necessary to introduce recursion:

```
billingS :: Pattern
billingS =
       next "c" ‘Seq‘ Bind (\[e1] ->
       next "s" ‘Seq‘ Bind (\[e2] ->
       Return [e1,e2] ‘Par‘ billingS))
```

With this new definition, every time a crosscut is found, a new one is looked for. Note that we used `Par`, which returns several results, rather than `Seq`. However, the crosscut instances must be sequential. Indeed, overlapping customer selections and service calls are not recognized as they should. For example, matching `billingS` on the trace `"ccss"` yields the single crosscut `[("c",1),("s",3)]` because after the first `"c"` has been matched, the pattern looks for the next `"s"` and skips the second `"c"`.

In order to detect the different instances of the billing crosscut in the presence of overlapping, the pattern definition must be modified to:

```
billingO ::  Pattern
billingO =
    next "c" ‘Seq‘ Bind (\[e1] ->
    billingO’ e1)

billingO’ ::  Event -> Pattern
billingO’ e1 =
    First (next "c" ‘Par‘ next "s") ‘Seq‘ Bind (\[e2] ->
    if (isName "s" e2)
      then Return [e1,e2]
      else billingO’ e1 ‘Par‘
           Filter (isName "s") (billingO’ e2))
```

In this case, we get the expected behavior: matching `billingO` on the trace `"ccss"` yields two crosscuts: `[("c",1),("s",3)]` and `[("c",2),("s",4)]`. The first definition `BillingO` matches the next `c` and passes it as a parameter to the auxiliary definition `billingO’`. This second crosscut definition matches either the next `"s"` or the next `"c"` whichever comes first. If `"s"` is matched, a crosscut has been found and it is returned. If `"c"` is matched, the beginning of a new instance has been detected and a new pattern must be matched in parallel. Note that, in this latter case, the next event `"s"` must not be taken into account by the newly created pattern (i.e. the second argument of `Par`). This is achieved with the help of the `Filter` construction. Hence, overlapping customer-service pairs are matched. (A crosscut definition `billingNS` matching nested pairs can be found at the end of Appendix A.)

Another crosscut definition is inspired by AspectJ [10]. In AspectJ, it is possible to define a *restricted* version of the billing crosscut as follows:

```
billingCflow = cflow(customer.select()) && execution(service.call())
```

This definition denotes crosscuts that relate each `service.call()` with the previous call to `customer.select()` that has not yet returned (i.e., the topmost call to `customer.select()` in the execution stack). Such a crosscut can be repeated and nested. For example, let `"c"` represents a call to `customer.select()`, `"r"` the corresponding return-statements and `"s"` a call to `service.call()`. Then, `cflow()` should yield three crosscuts for the trace `"ccssrsrs"`: `[("c",2), ("s",3)]`, `[("c",2), ("s",4)]` and `[("c",1), ("s",6)]`.

There is currently no formal definition of ASPECTJ's crosscut language. In fact, one of the initial motivations of our work was to be able to understand crosscutting in ASPECTJ without inspecting the code produced by the weaver. Using our framework, `cflow()` could be formally defined as follows (rather than having recourse to a new primitive pattern constructor):

```
billingCflow ::  Pattern
billingCflow =
  next "c"            ‘Seq‘ Bind (\[e1] ->
  billingCflow’ e1 billingCflow)

billingCflow’ ::  Event -> Pattern -> Pattern
billingCflow’ e1 k =
  First (next "c" ‘Par‘ next "r" ‘Par‘ next "s")
  ‘Seq‘ Bind (\[e2] ->
     if (isName "s" e2)
        then Return [e1,e2] ‘Par‘ (billingCflow’ e1 k)
        else if (isName "r" e2)
                then k
                else billingCflow’ e2 (billingCflow’ e1 k))
```

In the definition of `billingCflow’`, a continuation `k` is used in order to deal with nested crosscuts. Event `e1` represents the last customer selection that has not yet returned. Three cases have to be considered. First, if `"s"` is matched a crosscut starting at `e1` has been found, it is returned and further crosscuts — still relating to the customer selection `e1` — are searched for. If `"r"` is matched the continuation is used to restore the previous value of `e1`. Finally, if `"c"` is matched new crosscuts starting with the current selection `e2` have to be looked for and the value of `e1` is stored in the continuation. Note that this definition makes explicit the complex relationship (inherent to the `cflow()`-construction) between the points involved.

One advantage of such a formalization is that non-trivial properties can be *proven*, for instance, for optimization purposes. In our framework, a program execution (trace) can be modeled as a sequence of `Return`-statements. For example, the function

```
prog ::  Int -> Pattern
prog ts = prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
                      ‘Seq‘ Return [("s", ts+1)]
                      ‘Seq‘ Return [("c", ts)]
```

models a program repeatedly performing a customer selection and a service call before returning from the customer selection. Our formal definition allows the following property to be proven by means of the rules shown in Figure 3:

```
(prog ts) ‘Seq‘ billingCflow = (prog ts) ‘Seq‘ billingS
```

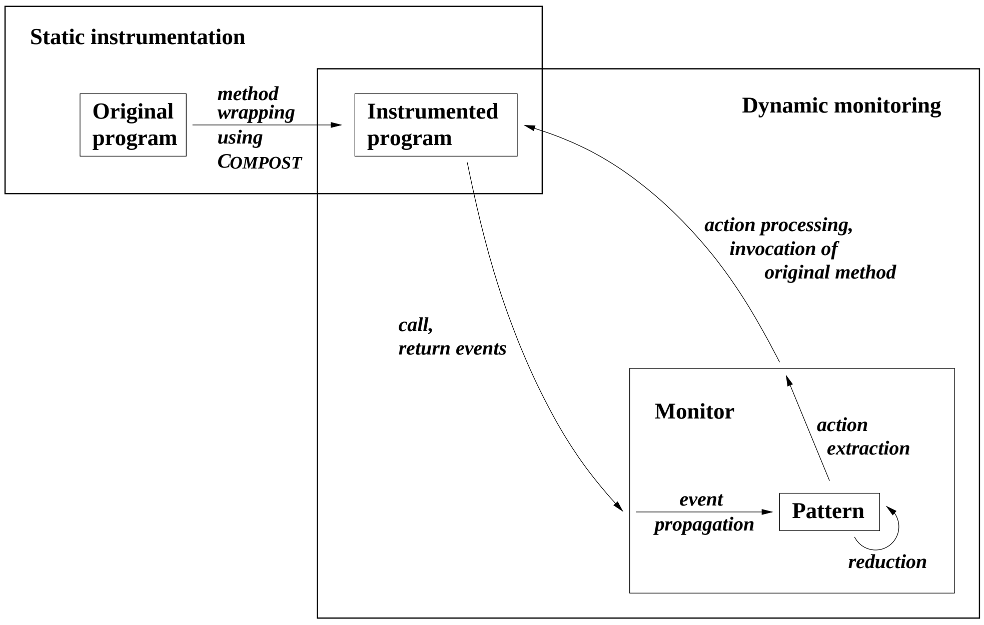

Figure 4: Architecture of the JAVA prototype

This property enables replacing a complex pattern (`billingCflow` whose continuation could, e.g., be implemented using a stack) by a simpler one (`billingS` which does not need a stack) in the context `prog`. The formal proof of this property can be found in Appendix B.

4 A JAVA prototype

This section describes a prototype (implemented itself in JAVA) of our framework for monitoring JAVA programs. We introduce the general architecture of the prototype first and detail then the two main parts of the architecture: static instrumentation and dynamic monitoring. The prototype has been implemented using two tools: COMPOST [1] for program instrumentation and PIZZA [15] for the translation of our HASKELL implementation into JAVA.

4.1 Architecture

The prototype implements the framework introduced in Section 3.1. Hence, it consists of two main parts: the program to be monitored and the monitor itself, both of which are implemented as JAVA programs. The two parts implement a producer-consumer pair and the whole prototype is single-threaded: the monitored program explicitly calls the monitor when an event is emitted, the monitor performs crosscut detection and hands back control to the monitored program. As shown in the upper left corner of Figure 4, this is implemented by statically instrumenting the original program with code constructing event objects and calls to the monitor. As shown in the lower right corner of Figure 4, the monitor reduces the current pattern representation at runtime and performs actions before handing back control to the monitored program. In the remainder of this section, we detail the two parts of the architecture.

4.2 Static instrumentation

In this section, we present the event representation and describe how we insert calls to the monitor into the monitored program by means of program transformation.

4.2.1 Event representation

Events model points in the execution of the monitored program. When an event object is created, information about the current program state must be stored in it. For instance, in our prototype, we focus on method calls by defining two event types (named `EvtCall` and `EvtReturn`). We associate the following information with such events:

- Caller
- Receiver
- Method name
- Arguments (only for `EvtCall` events)
- Result (only for `EvtReturn` events)
- Time stamp (enabling events to be ordered chronologically)
- Stack level
- Previous event object

Using this information, we may, for instance, easily relate an `EvtCall` with its associated `EvtReturn` by means of the method name and the stack level. Note that alternative representations exist which differ, for instance, with respect to the stored information.

4.2.2 Program instrumentation using Compost

As discussed previously, we currently use a single-threaded implementation method of the framework for monitor-based AOP which relies on calls to the monitor being inserted into the executing program. In our implementation, each method call of the original program is thus transformed as illustrated by the following piece of code:

```
public String method(String aString){
  String s;
  s = (String)Monitor.trace(new EvtCall(this, "method",new Object[] {aString}));
  s = (String)Monitor.trace(new EvtReturn(this, "method", s));
  return s;
}

public String methodOriginal(String aString){
  ... // original code
}
```

Note that the call to `Monitor.trace()` calls the constructors of the events with only three arguments. The remaining contextual information mentioned in Section 4.2.1 (such as the stack level) is added by the constructors. The monitor is responsible to call the original method when no crosscut is detected at this point. Otherwise, the monitor can perform an action, which may, for instance, replace the call to the original method by a call to another one.

In our current prototype, we use Compost [1] (version 0.61) — a general-purpose, source-to-source or bytecode-to-source transformation system — in order to automatically instrument programs with event-construction code and calls to the monitor. Compost reads a set of source and bytecode files, constructs an internal model of the complete program consisting of an abstract syntax tree and semantic relationships between tree nodes, and enables the application of transformations — defined as Java programs — to this model. Furthermore, Compost allows to ensure that transformations do not result in an incoherent program model (e.g., a change to method signature is propagated to the corresponding call sites).

Figure 5 shows the instrumentation algorithm using a pseudo-code formulation. This algorithm iterates over all method and constructor declarations of all compilation units which are to be transformed. Method and constructor declarations are transformed in two steps:

1. Rename the original method. This step requires primitive parameter types to be replaced by the corresponding object types (e.g., `int` becomes `Integer`) such that it can be called from the monitor using the Java Reflection API.

2. Create a (new) wrapper method having the original signature, constructing event objects for method calls and returns, and calling the monitor with these objects.

Note that even this quite basic transformation requires a transformational system which supports fine-grained transformations: apart from renaming methods and adding wrapper methods, we need to call the monitor from the executing program and vice versa. This means that the types of certain parameters must be changed and references to these parameters in method bodies must be modified accordingly.

```
main():
  for each compilation unit cu
    for each type definition td ∈ cu
      for each method and constructor declaration md ∈ td
        rename(md)
        create_wrapper(md, td)

rename(md):
  replace primitive parameter types by corresp. object types
  rename md to mdOriginal

create_wrapper(md, td)
  clone signature of md
  construct method body calling Monitor.trace() with appr. event objects
  add wrapper method to td
```

Figure 5: Instrumentation algorithm (pseudo-code)

```
TypeDeclarationMutableList tds = cu.getDeclarations();

for (int j=0; j<tds.size(); j++) {
  ClassDeclaration td = tds.getTypeDeclaration(j);
  MethodMutableList mds = td.getMethods();
  for (int k=0; k<mds.size(); k++) {
    MethodDeclaration md = mds.getMethod(k);
    transformMethod(md, td);
  }

  ConstructorMutableList cds = td.getConstructors();
  for (int k=0; k<cds.size(); k++) {
    Constructor c = cds.getConstructor(k);
    transformMethod(md, td);
  }
}

analyze();
transform();
```

Figure 6: `main()`: iterating over all method and constructor declarations

```
MethodDeclaration orgMD = md.deepClone();
// replace primitive types by corresponding object types in the signature
// modify body to translate changed parameters to original types
orgMD.setName(orgMD.getName() + "Original");
doReplace(md, orgMD);
```

Figure 7: `rename(md)`: renaming methods and constructors

```
MethodDeclaration wrapMD = MethodKit.cloneHeader(md);
Type rt = md.getReturnType();
if (rt == null) { // void method
  callArgument = // initialize primitive parameters as object types
  stBlock = "Monitor.trace(new EvtCall(this, \""
              + md.getName() + "\", " + callArgument + "));"
            + "Monitor.trace(new EvtReturn(this, \""
              + md.getName() + "\", null));";
} else ; // function: create body string with return value
StatementMutableList ss = getProgramFactory().parseStatements(stBlock);
StatementBlock body = getProgramFactory().createStatementBlock(ss);
wrapMD.setBody(body);
doAttach(wrapMD, td, td.getMembers().size());
```

Figure 8: `create_wrapper(md, td)`: creating wrapper methods

Figures 6–8 show most of the actual JAVA code defining the source-to-source transformation which implements the instrumentation.[7] Figure 6 shows the main inner loops iterating over all method and constructor declarations extracted from all type declarations in a compilation unit. After a method declaration has been singled out, a corresponding transformation is constructed by a call to `transformMethod(md, td)`. Once all declarations have been treated, the effects of the transformations on the overall program model are analyzed (`analyze()`) and, finally, the transformations are carried out (`transform()`).

Figure 7 presents the code constructing the renaming transformation. The original method is cloned, primitive types are replaced by their corresponding object types (i.e., by replacing arguments in the parameter section and references to these arguments in the body), the method name is modified, and, finally, a replacement transformation object is created (`doReplace()`).

Last but not least, Figure 8 shows the code creating the wrapper method. The header of the original method is cloned and a string `stBlock` representing the body of the wrapper method is constructed which consists of the two calls to the monitor (`Monitor.trace()`) having as arguments event objects for the method call (`EvtCall()`) and return (`EvtReturn()`) as introduced in Section 4.2.1. From this string, a statement block is created and linked to the wrapper method. Finally, a transformation object attaching the wrapper method to the current type declaration is constructed (`doAttach()`).

4.3 Dynamic monitoring

In this section, we describe the dynamic part of our prototype (cf. Figure 4). In the following subsections, we present how patterns are represented in JAVA, how they can be matched, and how actions are integrated in the prototype.

4.3.1 Data representation

In our prototype, the type `Event` of the HASKELL framework gives rise to a corresponding JAVA class encapsulating the context information. Similarly, a `Crosscut` encapsulates a list of events both in the HASKELL framework and the JAVA prototype.

The `Pattern` data type has been translated to a set of Java classes such that a pattern is represented by a tree of objects (an abstract syntax tree). The class `Pattern` is an abstract class defining a generic type for its subclasses: `Return`, `Bind`, `Seq`, `Filter`, `Abort`, `Par` and `First`. As illustrated in Figure 9 for expressions involving the operator `Par`, each of them implements three methods:

- `step()` which defines the reduction process of the pattern.
- `setActions(theActionTable)` which corresponds to `getReturns` in the HASKELL version,

[7]For presentational purposes, the code has been slightly simplified in two ways. Some part of the code has been replaced by comments describing its effects. Type casts which are frequently necessary to conform to the static semantics of JAVA have been omitted.

- `removeActions()` which corresponds to `removeReturns`.

4.3.2 Matching patterns

As presented in Figure 4, the Monitor is a component which essentially reduces a pattern until a fixpoint is reached. When a fixpoint is reached, the crosscuts (and associated actions) are extracted from the reduced pattern and the actions are executed. These actions possibly call the original method. If no crosscut is found, the original method is also called and the monitor hands control back to the monitored program.

As shown in Figure 10, the monitor implementation is derived from the Haskell source code. Event handling in the monitor is triggered by a call to the method `trace(Event)`.This method processes each event in turn. First, it propagates it by applying some of the rewrite laws to the pattern (`step()`) until a fixpoint is reached (`tmp2.equals(tmp1)`). Then the resulting actions are extracted from the transformed pattern (`setActions(theActionTable)`) and the resulting pattern is cleaned up (`removeActions()`) in order to prepare for the next trace analysis (`pattern = tmp2`). Finally, the actions and the original method invocation are processed (`invokeActions()`) and the result is returned to the caller (`return e.result`).

This monitor implementation allows us to define a billing crosscut using Pizza as follows:

```
static Pattern billingS() {
  return
    new Seq(Cafeine.next("Client","makeRequest",Cafeine.CALL),
            new Bind(
              fun(Crosscut client) -> Pattern {
              return new Seq(Cafeine.next("Service","serve",Cafeine.RETURN),
                             new Bind(
                               fun(Crosscut service) -> Pattern {
                               return new Par(new Action(
                                                new Crosscut(new Crosscut[]{client,service}),
                                                new Billing()),
                                              billingS());
                               }));
              }));
}
```

This definition is a direct translation of the Haskell code for `billingS` defined in Section 3.3. Note that this translation relies on the use of higher-order functions provided in form of Pizza's construct `fun(arg)->expr`.

4.3.3 Performing actions

Aspects are data structures relating crosscuts with actions. In our prototype, we decided to associate actions to patterns. More specifically, actions are declared using a special pattern

HASKELL code (excerpts the code shown in Appendix A):

```
step (Abort `Par` p2) = p2
step (p1 `Par` Abort) = p1
step (p1 `Par` p2)  = step p1 `Par` step p2

getReturns (p1 `Par` p2)    = getReturns p1 `Par` getReturns p2

removeReturns (p1 `Par` p2) = removeReturns p1 `Par` removeReturns p2
```

Derived JAVA code:

```
class Par extends Pattern {
  Pattern p1, p2;

  Pattern step() {
    if (p1 instanceof Abort) return p2;
    if (p2 instanceof Abort) return p1;
    return new Par(p1.step(), p2.step());
  }

  void setActions(Hashtable h){
    p1.setActions(h);
    p2.setActions(h);
  }

  Pattern removeActions(){
    return new Par(p1.removeActions(),p2.removeActions());
  }

  ...
}
```

Figure 9: Rewriting `Par`-expressions

HASKELL code:

```
data Monitor = M (Event -> ([Crosscut],Monitor)
monitor :: Pattern -> Monitor
monitor p = M (\event -> let p’ = reduce (Return [event] ‘Seq‘ p)

in (getReturns p’, monitor (removeReturns p’)))
```

JAVA code:

```
Object trace(Event e){
  current = e;

  //event propagation
  Pattern tmp1 = new Seq( new Return(current),pattern);
  Pattern tmp2 = tmp1.step();
  while (!tmp2.equals(tmp1)){
    tmp1 = tmp2;
    tmp2 = tmp1.step();
  }

  //action initialization
  initActionTable();
  tmp2.setActions(actionTable);
  tmp2 = tmp2.removeActions();
  pattern = tmp2;
  //action processing
  invokeActions();
  return e.result;
}
```

Figure 10: Execution monitoring

```
new Action(new Crosscut(new Crosscut[]{client, service}),
           new Billing())

public class Billing implements ActionCode{
       public void apply(Crosscut c){
            process_billing((Client)c.getEvent(0).getTo(),
                            (Service)c.getEvent(1).getTo());
       }
       (...)
}
```

Figure 11: `Action` constructor for the *billing* example

constructor `Action` which encapsulates an object with a method `void apply(Crosscut c)` implementing the action code (see Figure 11). When the reduction of the current pattern in the monitor reaches a fixpoint, the `Action` declarations are extracted from the `Pattern` by the `Monitor`. Once, all the extracted actions have been executed, the monitor invokes the original method (whose name is stored in the current `EvtCall` object) using the Java Reflection API and stores the result in the current event:

```
Method m = toClass.getMethod(currentEvent.methodId+"Original", argsClasses);
currentEvent.result = m.invoke(currentEvent.to, currentEvent.args);
```

In contrast to our restricted prototype, a general action language should obviously address the numerous underlying problems appropriately, for instance enable the execution of actions to be reorded.

5 Related Work

In this section, we briefly discuss the relationship of our work to other work on AOP, execution monitoring, monadic programming and reflection.

Formal approaches to AOP. Few work has been done on formal approaches to AOP. Notable exceptions are the frameworks presented by Lämmel [12], Fradet and Südholt [5] and De Meuter [13]. The first two articles discuss the introduction of aspects into program texts by means of source transformations. They are less general than our approach because they do not allow to relate arbitrary execution points. This limitation enables efficient woven code to be produced, in particular by taking into account static analysis techniques. The third article presents monadic programming as a framework for AOP which allows the introduction of new behavior in a structured manner but does not include any support for crosscut definitions.

AOP tools. ASPECT-J [10] provides a development environment for AOP in JAVA. In particular, it includes a language for crosscut definitions which enables points in the execution of JAVA programs (called "pointcuts") to be denoted. Pointcuts can be atomic such as "method call" or "field assignment" pointcuts but can also denote *sets* of execution points (as, e.g., the `cflow()` pointcuts). The language also includes a limited number of operators (most notably, set union and intersection) for the composition of pointcuts.

In contrast to our proposal, ASPECTJ's crosscut definition language is intimately tied to (the execution model of) JAVA and is limited in scope. In fact, the pointcut `cflow(customer.select())` denotes only points occurring between the call to `select()` and the corresponding return statement, which does not allow to define the billing crosscut as presented here. Furthermore, the understanding of ASPECTJ's crosscut language is hampered by lack of a precise definition.

HYPER/J [16] is another aspect weaver for JAVA providing substantial support for multiple domains with class- and method-granularity. However, it does not have a clear semantics and cannot relate execution events (e.g., there is no equivalent to `cflow()`).

Execution monitoring for security purposes. Event monitoring has been mainly used recently for security-relevant tasks, especially *intrusion detection* [11]. The language RUSSEL [14], for instance, is a turing-complete DSL describing algorithms on sequential streams of security-relevant events. It provides a rule-based language to define intrusion-detection based on conditions and associated actions. In contrast to our approach, such languages do not provide descriptive means to relate arbitrary events: RUSSEL definitions are formulated in a purely imperative style and it essentially relies on an ad-hoc mechanism for the definition of chained rule applications which is specifically-designed with intrusion detection in mind. Efficiency concerns are also a major issue with respect to using execution monitoring for security purposes. Hence, compilation techniques for merging monitors into monitored programs are an important research issue (see e.g. Colcombet and Fradet [2]).

Monadic parsing and postmortem trace analysis. Our DSL was strongly influenced by monadic parser libraries [7]. Indeed our pattern constructors are closely related to the monadic operators: our `Return` (resp. `Abort`, `Par`) corresponds to the monadic `result` (resp. `zero`, `plus`) and our pattern `p1 ‘Seq‘ Bind p2` corresponds to the monadic expression `p1 ‘bind‘ p2`.

The main difference comes from our interaction model: each time an event is emitted, the monitor must decide if a pattern has been detected. This lazy producer-consumer behavior is mandatory in order to be able to suspend the program execution to perform an action. So, multiple patterns composed with `Par` must be detected by proceeding in a lock-step manner. This is why we introduced an extra constructor `Bind` representing a closure (a partially matched pattern). The producer-consumer behavior also prohibits backtracking behavior. Hence, work on postmortem trace analysis as used for debugging in PROLOG [4] cannot be reused directly.

Reflection. Reflective systems, in particular those based on Meta-Object Protocols (MOP) [8], can be seen as programmable monitoring systems: the so-called hooks define points of interest and the metaobjects monitor the execution and perform actions. However, a MOP defines a fixed set of hooks and does not allow to denote arbitrary execution points (MetaJ [3] weakens this restriction by providing a technique to construct specially-tailored MOPs). Furthermore, since a metaobject is only concerned with "events" relating to the objects it is associated to, metaobjects do not have a completely global view of the execution. Finally, since formal treatment of MOP-based systems is rather difficult, they do not seem suitable as a model. They provide, however, a potentially valuable implementation platform.

6 Conclusion

In this report, we introduced a general and operational model for crosscut definitions based on execution monitors. Principally, we presented a formally-defined domain-specific language for crosscut definitions and we defined sophisticated crosscuts useful in realistic applications. This work also allowed us to propose a formal definition for AspectJ's crosscut constructor `cflow()` and enables formal proofs of optimization properties. Finally, we presented a Java prototype which has been derived from the formal DSL definition.

Future work. The work presented in this report provides numerous opportunities for future work.

First, we should experiment with our DSL by developing libraries of useful crosscuts. Alternative pattern languages (e.g. based on regular expressions) could be investigated. A Java-like syntax for pattern definitions should also be developed. Second, in order to guide the user, static analysis techniques dealing with cost measures (e.g. based on measures for pattern complexity) and interactions between aspects (e.g. patterns that overlap) should be designed. Third, for efficiency concerns, the monitor should be compiled into the monitored program and the elimination of superfluous events should be studied.

Finally, we focused on crosscut definitions in this report. As far as actions are concerned, a taxonomy of actions based on realistic applications should be built and a DSL for action composition should be provided to users (e.g. subsuming AspectJ's `before`, `after`, `around`, `dominate`, ...).

References

[1] Compost home page. `http://i44www.info.uni-karlsruhe.de/~compost`.

[2] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 54–66, N.Y., January 19–21 2000. ACM Press.

[3] R. Douence and M. Südholt. A generic reification technique for object-oriented reflective languages. *The Journal of Higher-Order and Symbolic Computation*, 14(1), 2001. Preprint version: see METAJ home page, `www.emn.fr/sudholt/research/metaj`.

[4] M. Ducasse. Opium: An extendable trace analyser for Prolog. *The Journal of Logic programming*, 1999.

[5] P. Fradet and M. Südholt. AOP: towards a generic framework using program transformation and analysis. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1998.

[6] P. Hudak, S. Peyton Jones, P. Wadler, et al. Report on the programming language HASKELL. *ACM SIGPLAN Notices*, 27(5), March 1992. HASKELL home page: `www.haskell.org`.

[7] G. Hutton and E. Meijer. Monadic parsing in HASKELL. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[8] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[9] G. Kiczales et al. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[10] G. Kiczales et al. An overview of ASPECTJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001. To appear, preprint version: see ASPECTJ home page, `www.aspectj.org`.

[11] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Computer Security Application Conference*, 1994.

[12] R. Lämmel. Declarative aspect-oriented programming. In *Partial Evaluation and Program Manipulation*, 1999.

[13] W. De Meuter. Monads as a theoretical foundation for AOP. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1997.

[14] A. Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection.* PhD thesis, Université de Namur, 1997.

[15] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.

[16] P. Tarr and H. L. Ossher. HYPER/J user and installation manual. Technical report, IBM Corp., 2000.

A Haskell implementation of a DSL for crosscutting

```haskell
type Event = (String, Int)
type Crosscut = [Event]

data Pattern =
          Return Crosscut
        | Bind (Crosscut -> Pattern)
        | Seq Pattern Pattern
        | Filter (Event -> Bool) Pattern
        | Abort
        | Par Pattern Pattern
        | First Pattern

step :: Pattern -> Pattern
step (Return e `Seq` (Bind f))        = f e                              -- R2
step (Abort `Seq` p2)                 = Abort                            -- R3
step (p1 `Seq` First p2)              = First (p1 `Seq` p2)              -- R9
step (First Abort)                    = Abort                            -- R10
step (First (Return e))               = Return e                         -- R11
step (First (Return e `Par` p2))      = Return e                         -- R12
step (First p)                        = First (step p)       -- propagate
step (Return [e] `Seq` Filter pred p2) =
       if (pred e) then p2                                               -- R13
                   else Filter pred (Return [e] `Seq` p2)                -- R13
step (Filter pred Abort)              = Abort                            -- R14
step (Filter pred (Return e))         = Return e                         -- R15
step (Filter pred (Return e `Par` p2)) = Return e `Par` Filter pred p2   -- R16
step (Filter pred p)                  = Filter pred (step p) -- propagate
step (p1 `Seq` (p2 `Par` p3))         = (p1 `Seq` p2) `Par` (p1 `Seq` p3) -- R6
step ((p1 `Par` p2) `Seq` p3)         = (p1 `Seq` p3) `Par` (p2 `Seq` p3) -- R7
step (p1 `Seq` (p2 `Seq` p3))         = step ((p1 `Seq` p2) `Seq` p3)    -- R1
step ((p1 `Par` p2) `Par` p3)         = p1 `Par` (p2 `Par` p3)           -- R4
step (Return e `Par` p2)              = Return e `Par` p2                -- R5
step (p1 `Par` Return e)              = Return e `Par` p1                -- R5
step (p1 `Par` (Return e `Par` p2))   = Return e `Par` (p1 `Par` p2)     -- R5
step (Abort `Par` p2) = p2                                               -- R8
step (p1 `Par` Abort) = p1                                               -- R8
step (p1 `Seq` p2)                    = step p1 `Seq` step p2
step (p1 `Par` p2)                    = step p1 `Par` step p2
step (Return e)                       = Return e
step (Bind f)                         = Bind f
step (Abort)                          = Abort
```

```
run :: Program -> Monitor -> [Crosscut]
run Over           _           = []
run (Cont program) (M monitor) = let (event,program') = program ()
                                     (crosscuts,monitor') = monitor event
                                  in crosscuts ++ (run program' monitor')

run' :: Pattern -> String -> [Crosscut]
run' p s = run (program s) (monitor p)

data Program = Over | Cont (() -> (Event,Program))

program :: String -> Program
program s = program' s 1

program' :: String -> Int -> Program
program' ""     _ = Over
program' (c:cs) n = Cont (\() -> (([c],n), program' cs (n+1)))

data Monitor = M (Event -> ([Crosscut],Monitor))

monitor :: Pattern -> Monitor
monitor p = M (\event -> let p' = reduce (Return [event] `Seq` p)
                         in (getReturns p', monitor (removeReturns p')))

reduce :: Pattern -> Pattern
reduce p = fixPoint step p
           where fixPoint f x = if (f x == x) then x else fixPoint f (f x)

removeReturns :: Pattern -> Pattern
removeReturns (Return _)        = loop where loop = Bind (\e -> loop)
removeReturns (p1 `Par` p2)     = removeReturns p1 `Par` removeReturns p2
removeReturns (p)               = p

getReturns :: Pattern -> [Crosscut]
getReturns (Return e)           = [e]
getReturns (p1 `Par` p2)        = getReturns p1 ++ getReturns p2
getReturns _                    = []

next :: String -> Pattern
next s = Bind (\[e]-> if (isName s e) then Return [e] else next s)
```

```
isName :: String -> Event -> Bool
isName n (n',_) = n==n'

billingNSTest = run' billingNS "ccscsscs"
-- [[("c",2),("s",3)],[("c",4),("s",5)],[("c",1),("s",6)],[("c",7),("s",8)]]

billingNS = next "c"               `Seq` Bind (\[e1] ->
        billingNS' e1 billingNS)

billingNS' e1 k =
        First (next "c" `Par` next "s") `Seq` Bind (\[e2] ->
        if (isName "s" e2)
                then Return [e1,e2] `Par` k
                else  billingNS' e2 (billingNS' e1 k))
```

B Proof of an optimization property involving `cflow`

As discussed at the end of Section 3.3, our formal definition of crosscuts enables properties to be formally proved. Remember, a program can be modeled in our framework as a pattern:

```
prog :: Int -> Pattern
prog ts = prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
                      ‘Seq‘ Return [("s", ts+1)]
                      ‘Seq‘ Return [("c", ts)]
```

In this context, `billingCflow` is equivalent to `billingS`:

```
(prog ts) ‘Seq‘ billingCflow = (prog ts) ‘Seq‘ billingS
```

This property can be easily proven by means of definition folding/unfolding and the rules shown in Figure 3. The main steps of this proof are:

```
(prog ts) ‘Seq‘ billingCflow
= (unfold prog)
  (prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
               ‘Seq‘ Return [("s", ts+1)]
               ‘Seq‘ Return [("c", ts)]) ‘Seq‘ billingCflow
= (R1)
  prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
              ‘Seq‘ Return [("s", ts+1)]
              ‘Seq‘ Return [("c", ts)] ‘Seq‘ billingCflow
= (unfold billingCflow and R1)
  prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
              ‘Seq‘ Return [("s", ts+1)]
              ‘Seq‘ Return [("c", ts)] ‘Seq‘ next "c" ‘Seq‘ Bind (\[e1] -> billingCflow’ e1 billingCflow)
= (definition of next and R2)
  prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
              ‘Seq‘ Return [("s", ts+1)] ‘Seq‘ billingCflow’ ("c", ts) billingCflow
= (unfold billingCflow’ and R1)
  prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
              ‘Seq‘ Return [("s", ts+1)] ‘Seq‘
              First (next "c" ‘Par‘ next "r" ‘Par‘ next "s")
                ‘Seq‘ Bind (\[e2] ->
                  if (isName "s" e2)
                      then Return [("c", ts),e2]
                           ‘Par‘ (billingCflow’ ("c", ts) billingCflow)
                      else if (isName "r" e2)
                              then billingCflow
                              else billingCflow’ e2 (billingCflow’ ("c", ts) billingCflow))
= (definition of next and R11, R12, R2)
  prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
              ‘Seq‘ if (isName "s" ("s", ts+1))
                       then Return [("c", ts),("s", ts+1)]
                            ‘Par‘ (billingCflow’ ("c", ts) billingCflow)
                       else if (isName "r" ("s", ts+1))
                               then billingCflow
                               else billingCflow’ ("s", ts+1) (billingCflow’ ("c", ts) billingCflow)
= (definition of isName)
  prog (ts+3) ‘Seq‘ Return [("r", ts+2)]
              ‘Seq‘ (Return [("c", ts),("s", ts+1)]
                     ‘Par‘ (billingCflow’ ("c", ts) billingCflow))
```

```
= (unfold billingCflow')
  prog (ts+3) `Seq` Return [("r", ts+2)]
              `Seq` (Return [("c", ts),("s", ts+1)]
              `Par` First (next "c" `Par` next "r" `Par` next "s")
                           `Seq` Bind (\[e2] ->
                              if (isName "s" e2)
                                 then Return [("c", ts),e2]
                                      `Par` (billingCflow' ("c", ts) billingCflow)
                                 else if (isName "r" e2)
                                        then billingCflow
                                        else billingCflow' e2 (billingCflow' ("c", ts) billingCflow)))
= (R6)
  prog (ts+3) `Seq` ((Return [("r", ts+2)]
                      `Seq` Return [("c", ts),("s", ts+1)])
                    `Par`
                     (Return [("r", ts+2)]
                      `Seq` First (next "c" `Par` next "r" `Par` next "s")
                            `Seq` Bind (\[e2] ->
                               if (isName "s" e2)
                                  then Return [("c", ts),e2]
                                       `Par` (billingCflow' ("c", ts) billingCflow)
                                  else if (isName "r" e2)
                                         then billingCflow
                                         else billingCflow' e2 (billingCflow' ("c", ts) billingCflow))))
= (definition of next and R11, R12, R2)
  prog (ts+3) `Seq` ((Return [("r", ts+2)]
                      `Seq` Return [("c", ts),("s", ts+1)])
                     `Par`
                     (if (isName "s" ("r", ts+2))
                         then Return [("c", ts),("r", ts+2)]
                              `Par` (billingCflow' ("c", ts) billingCflow)
                         else if (isName "r" ("r", ts+2))
                                 then billingCflow
                                 else billingCflow' ("r", ts+2) (billingCflow' ("c", ts) billingCflow)))
= (definition of isName)
  prog (ts+3) `Seq` ((Return [("r", ts+2)]
                      `Seq` Return [("c", ts),("s", ts+1)])
                    `Par`
                    billingCflow)
```

In the same way, it is easy to show that

```
(prog ts) `Seq` billingS =
  prog (ts+3) `Seq` ((Return [("r", ts+2)]
                     `Seq` Return [("c", ts),("s", ts+1)])
                    `Par`
                    billingS)
```

So, the property holds.