The abc Group

Relational Aspects as Tracematches

abc Technical Report No. abc-2007-4

Eric Bodden, Reehan Shaikh and Laurie Hendren
School of Computer Science
McGill University
Montréal, Québec, Canada

February 8, 2008

a s p e c t b e n c h . o r g

Contents

List of Figures

List of Tables

Abstract

The relationships between objects in an object-oriented program are an essential property of the program's design and implementation. Two previous approaches to implement relationships with aspects were *association aspects*, an AspectJ-based language extension, and the *relationship aspects* library. While those approaches greatly ease software development, we believe that they are not general enough. For instance, the library approach only works for binary relationships, while the language extension does not allow for the association of primitive values or values from non-weavable classes. Hence, in this work we propose a generalized alternative implementation via a direct reduction to tracematches, a language feature for executing an advice after having matched a sequence of events. This new implementation scheme yields multiple benefits. Firstly, our implementation is more general than existing ones, avoiding most previous limitations. It also yields a new language construct, relational tracematches. We provide an efficient implementation based on the AspectBench Compiler, along with test cases and microbenchmarks. Our empirical studies showed that our implementation, when compared to previous approaches, uses a similar memory footprint with no leaking, but the generality of our approach does lead to some runtime overhead. We believe that our implementation can provide a solid foundation for future research.

1 Introduction

The relationships between objects are an important property of any object-oriented program and software architecture, regardless of whether or not aspect-oriented programming is used. These relationships exist naturally. They become apparent at the latest in the form of design patterns [11] or architectural styles [12]. In most programming languages however, such relationships cannot be explicitly expressed. They rather have to be encoded via references between the involved objects. This has a fundamental drawback: while a relationship between objects can (and usually will) have *semantics* attached, a simple reference does not. This semantics hence need to be implemented elsewhere, most commonly in the classes of the participating objects. This may lead to both scattering and tangling [22] of the source code for the given relationship.

Researchers have therefore proposed to implement such high-level relationships with aspects [14]. Yet, implementations resorting to plain AspectJ exhibit one problem. Although it is relatively easy to implement the *behaviour* of a relationship via advice, programmers have to keep track of the *state* associated with those relationships manually. This results in a lot of redundant and boilerplate code that distracts from the actual core logic which the relationship is meant to implement.

Two approaches, *association aspects* [18, 19] and the *relationship aspects library* [16, 17], hence try to improve on this situation, via different approaches. Association aspects implement a language extension to AspectJ which generates the necessary boilerplate code automatically. Relationship aspects on the other hand offer a library of generic abstract aspects that provide default implementations for some of the most commonly used relationships.

While we believe that those implementations do ease software development, they still carry certain limitations. Association aspects, for instance, do not allow the programmer to associate values of a primitive type or objects of non-weavable classes. It is common practice to *not* weave into the Java runtime library. This implies that no objects of types from this library can be used in associations. However, many of those types represent data values (e.g. String, Integer, Date, . . .) and occur *naturally* in associations.

The relationship aspects library, on the other hand, only supports binary relationships. Along with this work we expose several examples that relate more than two objects with each other. Hence, we are interested in a more general solution.

Another approach to abstract from object relations exists, however for an entirely different purpose. *Tracematches* [1] allow a programmer to reason about sequences of events which occur during program execution and involve a given group of objects. For example, a tracematch may automatically raise an error when an iterator for some collection is advanced although the collection has been updated after the iterator's creation [8]. Our own background lies in the design and implementation of a static whole-program analysis [8] to increase the runtime performance of tracematches.

The fact that both, tracematches and language support for implementing relations via aspects, have to deal with the same problem of efficiently associating related state, made us think whether it would not be possible to implement the one approach using the other and whether or not synergistic effects would arise when doing so. In this work we show that we can in fact directly implement a variant of association aspects, coined *relational aspects*, using tracematches whilst incorporating all of the desired features mentioned in previous related work. We present such an implementation and furthermore show that this implementation scheme solves most known limitations of the previous approaches.

Moreover, the careful design of tracematches automatically guarantees for the implementation's memory-safety and for fast value look-up through optimized indexing. Finally, the implementation is easily seen to be correct, assuming a correct implementation of tracematches.

On the other hand, tracematches can gain through the availability of relational aspects. Their combination yields an entirely new language feature, *relational tracematches*. A relational tracematch is matched against sequences of events but only taking into account those events that involve objects that have been associated with the relationship the aspect represents.

Along with this work, we expose a full implementation of relational aspects and relational tracematches using the AspectBench Compiler [3], including a variety of test cases and microbenchmarks. The test cases validate the correctness of the implementation and demonstrate use cases for relational aspects and relational tracematches. The benchmarks help us to estimate the cost at which our flexible solution comes. Further, they revealed interesting insights about the importance of finding an efficient yet flexible storage structure. As our results show, our implementation is memory-safe. While it is less efficient than those presented in related work, its runtime overhead is still very reasonable. Future optimizations planned for tracematches promise to increase the efficiency even to the same level as for the other approaches.

Contributions To summarize, in this work we present the following original contributions:

1. a detailed description of the correspondence between the two previously existing language features of association aspects and tracematches, and the relationship aspects library,
2. an extension to the AspectBench Compiler implementing relational aspects via tracematches,
3. a full account of the important features that come with this implementation scheme, and
4. the first performance study investigating the relative performance of different approaches in the field, and ours.

We organized the remainder of this paper as follows. In Section 2 we first discuss related work and show how it motivates our own approach, relational aspects. The syntax and semantics of relational aspects are given in Section 3, while Section 4 describes in detail their implementation via a reduction to tracematches. As mentioned earlier, our implementation exposes many useful features and overcomes shortcomings of earlier approaches. We discuss this in detail in Section 5. One particularly interesting feature is the support of a new language construct, relational tracematches. Section 6 discusses their syntax, semantics and applications. In Section 7 we conduct a performance evaluation comparing related work with ours. We conclude in Section 8.

2 Related work

We decided to categorize our related work by the way in which they implement a simple inter-object relationship, the Observer pattern [11]. This design exemplifies the case where one object is temporarily related with some others. Specifically, *observers* can register with a *subject* to be notified whenever the observable state of the subject changes. The observers in turn can then update their internal representation of the subject accordingly.

2.1 Object-oriented solution by Gamma et al.

The gang-of-four [11] suggested two possible implementations of this pattern in an object-oriented programming language in 1995. Firstly, each subject could store a list of observers that are currently registered with it. Whenever an operation changing the subject's observable state is invoked, all those observers are notified. If many possible subjects exist but only few of them are observed, it might however be too costly to store a list per subject. Hence, a second possible implementation was proposed, storing subject/observer associations using a hash table.

Both implementations share the problem that the actual business logic of each subject (which is certainly *not* to update its observers) is polluted with code implementing the Observer pattern. While in part this problem can be solved by having subjects inherit from an abstract Subject class, in languages with single inheritance this might not be an option.

2.2 AspectJ solution by Hannemann and Kiczales

In 2002, Hannemann and Kiczales [14] demonstrated that this particular design pattern can actually be implemented in a modular way using one single aspect in the aspect-oriented programming language AspectJ. This implementation eases reasoning about the relationship between registered subjects and observers by collocating all relevant code in one single unit. However, due to the lack of support for explicitly denoting relations and associations in AspectJ, the aspect still has to keep track of related objects manually. Hannemann and Kiczales used a hash map for this purpose.

One can argue that from a software-engineering perspective it is desirable to denote relationships between objects implemented by aspects rather explicitly, eliminating the burden of manual bookkeeping of such relations. As outlined below, two such approaches have previously been suggested.

2.3 Association aspects by Sakurai et al.

In 2004, Sakurai et al. proposed association aspects [18, 19], a language extension to AspectJ, allowing programmers to associate objects *explicitly* via an aspect. For that purpose, the signature of an aspect was extended. While normally AspectJ allows only for per-this, per-target, per-cflow and per-type-within instantiation of aspects, association aspects allow a programmer to associate an arbitrary vector of objects with each other and an aspect instance.

```
abstract aspect TimedObserver perobjects(Subject, Observer) {
  abstract pointcut subjectChanged(Subject s);
  long lastNotify;

  TimedObserver(Subject s, Observer o) {
    associate(s, o);
  }

  after(Subject s, Observer o) :
    subjectChanged(s) && associated(s,o) {
    long delta = System.currentTimeMillis() − lastNotify;
    if(delta>10000) {
      o.notify(s);
      lastNotify = System.currentTimeMillis();
    }
  }
}
```

Figure 1: Observer pattern as association aspect

Figure 1 shows one implementation of the Observer pattern in an association aspect. In this example, each observer is to be notified about the update to each associated subject at most once every 10 seconds.

In line 1, the aspect TimedObserver declares that it relates a Subject to an Observer. In line 2 it declares an abstract pointcut that will be triggered on any state change to a subject, exposing the subject itself. In line 3, we store a long value that is supposed to hold the time the last notification took place. Lines 5-7 declare an aspect constructor. Programmers can explicitly call this constructor in AspectJ code. The constructor invokes the auto-generated method associate (..) , which associates the constructed aspect instance with the subject and observer. Then in lines 9-16 the aspect declares a piece of advice that is executed whenever the subject s is changed *but only* if s is associated with an observer o. The advice then notifies the observer o about the state change in s, but only if the last notification of *this very observer* o about an update to *this very subject* s was more than 10 seconds ago. To be clear, this means that the field lastNotify is stored *per association.*

In order to associate a concrete subject s1 with an observer o1, client code calls **new** ObserverAspect(s1,o1). The constructor then establishes the association via the call to associate (..) .

Association aspects are implemented via an extension to the *ajc* compiler[1] for AspectJ. The compiler reduces association aspects to normal aspects, augmented with additional code to keep track of those relationships.

We believe that association aspects implement this Observer pattern very nicely. Consequently, the implementation we propose is very similar in flavour. The contribution of our work is not to improve on the syntax or semantics of association aspects but rather to demonstrate how a language feature like association aspects can be more flexibly implemented using tracematches.

This is because association aspects still suffer from one particularly severe limitation. They store associations directly via references introduced to the associated objects. This limits the approach to weavable classes only. It is not possible to relate objects of non-weavable classes, e.g. Strings or any other class of the Java runtime library which is not normally woven into. Association of primitive values is also not possible. As we will later show, our tracematch-based implementation does not suffer from such limitations.

2.4 Relationship aspects library by Pearce and Noble

While Sakurai et al. opted for a compilation-based approach to implementing relations via aspects, in 2006 Pearce and Noble [16, 17] addressed the same problem using a library of generic abstract aspects, the relationship aspects library. It is written in AspectJ5 which supports generic types as defined for Java5 [13].

Pearce and Noble demonstrated very convincingly how such a library can ease and promote the use of such a technology in actual AspectJ programs. For instance, apart from "standard" directed binary relations, their library provides symmetric relationships. We believe that no matter what implementation technique is used to provide relations via aspects in the back-end, such generic aspects can be useful in their own right, on top of any such implementation.

Unfortunately, however, some limitations of AspectJ prohibit the general applicability of their approach. For instance, their SimpleStaticRel aspect, an aspect designed for static relationships where objects are meant to be associated with each other for longer periods of time, uses inter-type declarations to store associations between objects. If now multiple relationship types, both sub-aspects of SimpleStaticRel, apply to the same element type, those inter-type declarations will lead to name clashes, triggering a bug in the *ajc* compiler[2]. Furthermore, their library only supports binary relationships, which to us is a potentially severe limitation that cannot easily be overcome. To allow up to n-ary relations, one would have to implement at least *o(n)* different generic aspects in their library. A specialized compiler like the one for association aspects can generate such code automatically, taking care to avoid name clashes as well. As we show later on, our tracematch-based approach does not suffer from these kinds of problems.

[1] ajc compiler: `http://www.eclipse.org/aspectj/`

[2] See bug #120015 at `https://bugs.eclipse.org/bugs/` for details.

```
1  abstract pointcut subjectChanged(Subject s);
2
3  tracematch(Subject s, Observer o) {
4    sym register_observer after returning:
5      call (* Subject.register (Observer)) && target(s) && args (o);
6    sym update_subject after:
7      subjectChanged(s);
8
9    register_observer  update_subject+ {
10     o.notify(s);
11   }
12 }
```

Figure 2: Tracematch implementing the Observer design pattern

2.5 Tracematches

In 2005, Allan et al. [1] proposed an AspectJ language extension called tracematches, but for a purpose other than associations. Tracematches do not abstract over relationships, but rather over the execution history of a running AspectJ program. They are implemented using the AspectBench Compiler [3].

Figure 2 shows a tracematch implementing the Observer pattern. For simplicity, timing information is left out. In line 1, we first specify the same abstract pointcut for updates to subjects as before. Line 3 then starts the actual tracematch declaration, by first specifying that the tracematch is going to reason about two objects, a Subject s and an Observer o. Lines 4-7 then set up an alphabet of “symbols”, where each symbol matches an AspectJ joinpoint. The symbol register_observer matches whenever any Observer o is registered with any Subject s. The symbol update_subject in turn matches whenever the Subject s is changed, as specified through the abstract pointcut. Lines 9-11 then finally hold the so-called tracematch pattern and the body. The pattern is a regular expression over the alphabet of symbols we just defined. Here, we wish to match whenever any specific Observer o has been registered with a Subject s and afterwards at least one update to this subject has been seen. The regular expression (line 9) implements this. In the back-end, the AspectBench Compiler generates a state machine keeping track of the internal tracematch state, in particular of partial matches. If multiple observers are registered with the same Subject s, a match will occur for all those observers. The tracematch runtime will execute the tracematch body for any such match, with s and o bound to the respective objects. The body so notifies the observer of the change in the subject.

Looking at this tracematch specification, at first it seems very different in style compared to the association aspect from Figure 1. While a tracematch specification has a regular expression and symbols, an association aspect does not. On the other hand, while an association aspect is explicitly being associated with a certain combination of objects, in a tracematch this association occurs implicitly, through matching symbols against a stream of events.

Nevertheless, we noted certain important similarities as well: both association aspects and tracematches relate a vector of objects among one another. In both models, there is a certain event that triggers a body of code being executed with variables bound to this vector of objects. Further, in our particular example, in the case of association aspects we only wish to execute the body for updates on subjects with which an observer has previously registered. In the tracematch, we model this behaviour via prefixing the regular pattern with register_observer .

Those similarities made us wonder whether or not tracematches actually subsume association aspects and in particular, whether association aspects could not be implemented via a reduction to tracematches. In the remainder of this paper we will demonstrate such an implementation and in particular we will describe how it avoids the aforementioned limitations of previous approaches.

2.6 Other related work

Here we briefly discuss other related work that did not directly influence our approach but motivates its importance.

2.6.1 Declarative Object Identity Using Relation Types

Recently, Vaziri et al. [23] reported on the problem of correctly implementing object identity via the methods equals (..) and hashCode() in Java. As they show in their case study, those methods are hard, if not sometimes impossible, to implement correctly. As a consequence, they suggest a language feature called *relation types* that encodes an equality relationship explicitly and in its own unit of code. The authors suggest a syntax and semantics very thoroughly tailored to the special purpose of providing a notion of *equality*. Yet, we believe that in general this problem could be solved as a special instance of a relational aspect, although probably not quite as concise. In any case, [23] strongly supports the claim that inter-object relationships are important in object-oriented programs, equality being one such relationship of special importance.

2.6.2 A relational model of object collaborations

Concurrently, Balzer et al. [6] described a relational model of object collaborations and its use in reasoning about relationships. The authors do not describe an implementation language for relationships but rather a *specification* language that can be used to enforce constraints over those relations. The constraints heavily rely on *member interposition* through relations. Interestingly, their "interposed members" are exactly equivalent in semantics to inter-type declarations by (potentially relational) aspects, while their "non-interposed members" are exactly equivalent to the aforementioned per-association state. Future work could decide whether their specification formalism can be used to verify constraints over the relational aspects proposed here.

2.6.3 Dynamic aspect implementation

The aspect-oriented programming community has developed implementations of aspect-oriented programming languages that are more dynamic than AspectJ. JAsCo [21] and CaesarJ [2] are only two examples of such languages. Composition filters [7] describe a model and language for the dynamic enablement and composition of aspects. Such dynamic approaches show their strength in providing relatively flexible forms of aspect deployment and configuration.

3 Syntax and semantics of relational aspects and relational advice

Our syntax and semantics for relational aspects were strongly inspired by the work on association aspects by Sakurai et al.. Nevertheless, in our approach, we opted for a syntax that is slightly closer to tracematches, for practical reasons.

3.1 Design decisions with respect to association aspects

Association aspects as proposed by Sakurai et al. introduced the following syntactic and semantic extensions to AspectJ:

1. Aspect declarations were enhanced to accept a vector of types: A declaration **aspect** ObserverAspect ... can be extended to aspect ObserverAspect(Subject,Observer) ... .
2. A new pointcut associated(x_1 ,.., x_n) was introduced. This allows to bind the associated objects to names.

3. Constructor invocations on aspects were allowed to create an aspect instance and potentially associate the instance with a vector of objects: **new** ObserverAspect(s1,o1);

4. An aspect instance is given an implicitly declared delete() method, that revokes the related association.

For relational aspects we chose a style closer to tracematches. In particular, extension (1.) was altered, so that the aspect header not only takes a list of types but a list of formal parameters, i.e. combinations of types and *names*. This comes closer to the syntax and semantics of tracematches, where we have a header that takes formal parameters which are bound over the lifetime of the tracematch (cf. line 3 of Figure 2).

This way each formal parameter is given a unique name and the **associated**-pointcut in (2.) becomes mostly superfluous (see Section 5.8 for details).

Last but not least, we didn't support the idea of allowing programmers to explicitly call an aspect's constructor. This is because in general, AspectJ does not allow to explicitly instantiate aspects. The syntax and semantics of association aspects break with that convention. We instead opted for a slightly different approach, using two auto-generated static methods associate (..) and release (..), as discussed below. These respectively replace the explicit constructor calls (3.) and the delete() instance method (4.).

In general, however, we wish to emphasize that the focus of this paper is *not* to discuss the best possible syntax and semantics for association aspects but their relationship to tracematches.

3.2 Syntax of relational aspects

In summary, relational aspects extend the AspectJ syntax only by two single grammar productions:

```
extend modifier ::= "relational";

extend aspect_declaration ::=
  modifiers_opt "aspect" "(" formal_parameter_list ")"
  super_opt interfaces_opt aspect_body;
```

The only newly added syntactic features are the **relational** modifier and the formal parameter list in the aspect declaration. Based on that definition, our parser accepts the relational aspect in Figure 3 as syntactically correct.

```
relational abstract aspect SimpleObserver(Subject s, Observer o) {

  abstract pointcut subjectChanged(Subject subj);

  relational after(): subjectChanged(s) {
    o.notify(s);
  }
}
```

Figure 3: Observer pattern as relational aspect

3.3 Static semantics of relational aspects

Again, this relational aspect implements the Observer pattern. The header hence takes a subject and an observer as arguments. In contrast to the syntax of association aspects, a relational aspect receives these arguments directly in the header, as in tracematches. These aspect parameters may be accessed from any *relational* advice declaration inside the aspect (and their pointcuts), as if those parameters were bound variables. (In fact, as the operational semantics will show, we assure that they *will* be bound when evaluated.) In the example, the programmer accesses the subject s in the pointcut of the advice. The advice body accesses both s and any associated observer o. If the programmer needs to access parameter values from within methods in the aspect, she has to explicitly expose these values to the method, either by passing them to the method via parameters or by storing them into fields. This scheme allows for automatic garbage collection of associated values in cases where their values are *not* stored by the user (see Section 5.2 for details).

We extended the type checker to make sure that the keyword **relational** only occurs in front of aspect declarations and advice declarations. In addition, we check the following: Relational advice may only occur inside relational aspects. Parameters may only be given to aspects that are flagged as relational. The parameter list for a relational aspect may be empty (see Section 5.11 for details). If a relational aspect extends another aspect, that aspect must also be relational and accept the same parameter types.

Pieces of advice that are not prefixed with the **relational** modifier use the default semantics for AspectJ. They may hence not access any aspect parameters.

For any relational aspect RA with parameters (T1 p1 ,..., Tn pn) the compiler declares public static methods RA.associate(T1 ,..., Tn) and RA.release(T1 ,..., Tn). The first one associates a new vector of objects while the second one releases it.

The method **aspectOf**(), as it is usually available for aspects is not available for relational aspects. This raises the question how a programmer should enumerate all objects bound to a relational aspect and the aspect instance related to theses objects. Interestingly, the use cases we found so far seem to suggest that in practice there is no need for such enumeration. Dynamic dispatch on the associated values, as implemented through relational advice, seems far more important and seems sufficient. Nevertheless, programmers can opt to simulate explicit look-up methods by implementing special relational advice. We expose an abstract aspect that implements object look-up this way, along with our implementation.

3.4 Operational semantics of relational aspects

The most interesting question is when exactly a relational advice executes, and if so, under which variable bindings. Variable names are disambiguated as follows: If a relational advice refers to a name n and there exists an aspect parameter with the same name, the name represents any value stored in that parameter. By *any* we mean that if multiple objects have been associated with that parameter, the advice body will execute *for each* such association. Note how this is in sync with the tracematch semantics. If a field of the same name exists, the programmer has to access this field via explicit qualification with **this**.

We do not forbid the association of the value **null**. However, its association will have no effect. To us it would have no meaning to relate anything to the **null** value.

Release As mentioned earlier, the programmer further has the possibility to release an association by calling the release (..) method. If this method is called on a vector v of objects, the association for v (and all associated aspect state) is dropped. Objects in v can be associated with the same aspect again by calling associate once again.

Instance fields As in association aspects, we define that instance fields of the aspect exist *per association.* This means that for every object vector v associated with a relational aspect, this aspect will have a copy of each field for each such v. Static fields on the other hand are unique, because they are members of the underlying class.

This concludes our description of the semantics of relational aspects. Let us now get to the crux of this paper, where we describe how this semantics can quite easily be implemented via a reduction to tracematches.

4 Implementation via tracematches

In the semantics section we noted that a vector of objects v is associated with a relational aspect RA if associate (v) has been called one or more times, and the last such call was not followed by a call to release (v). For somebody familiar with tracematches, this immediately reads like a tracematch pattern, because it can be described by a regular expression over the program's execution history.

4.1 A first, simple translation

In the following we give a first, simple translation that is already *almost* complete. The only feature missing will be the one of per-association fields. We will get back to this feature in the subsequent section.

Symbol definitions Let us assume that we are given a relational after-advice with a pointcut pc (..) . Then we can define a symbol "action" as follows:

sym action **after**: pc (..);

In addition, we define two more symbols, associate and release, that match calls to the respective methods of the relational aspect.

sym associate **after**: **call**(* RA.associate (..)) && **args**(x,y);
sym release **after**: **call**(* RA.release (..)) && **args**(x,y);

Here x,y is the vector of variable names induced by the parameter definition in the header of the relational aspect.

Regular expression Those three symbols define the alphabet {action, associate, release} for the regular expression of the tracematch. We claim that the following regular expression over this alphabet implements our desired semantics for relational advice.

associate action+

This is because the pattern matches whenever the original pointcut of the relational advice would have matched (via the action symbol), but only if a call to associate was seen before, with no call to release in between. Note that also traces like the following are matched, where release occurs *before* associate:

action associate release associate associate action

This is because the tracematch semantics define that the regular pattern is matched against each *suffix* of the execution trace (see [1] for details). Here, the suffix "associate action" is matched by the pattern, hence we match after the last action.

Tracematch variables The tracematch formal parameters are the same as the ones originally given to the relational aspect.

Generic Translation Figure 4 shows the tracematch generated from the simple observer in Figure 3. We describe the generic translation process while referring to the above example, thus allowing the reader to get a concrete sense of the process itself.

First, the compiler executes the following steps for each single relational advice.

1. The compiler generates an empty tracematch with the same formal parameters (line 4) as the surrounding relational aspect declaration.
2. It then adds the generic definitions for the two symbols associate and release (lines 5-8), where the **args**-pointcut holds the names of the tracematch parameters.
3. The symbol action (line 9) is added with the appropriate advice specification from the original advice and with the original pointcut (in our example, the **after**-advice).
4. Further, the compiler adds the generic pattern "associate action+" as well as the original advice body, which now becomes the tracematch body (lines 11-13).

```
1  aspect SimpleObserver{
2    abstract pointcut subjectChanged(Subject subj);
3
4    tracematch(Subject s, Observer o) {
5      sym associate after:
6        call(* SimpleObserver.associate (..))  && args(s,o);
7      sym release after:
8        call(* SimpleObserver.release (..))  && args(s,o);
9      sym action after: subjectChanged(s);
10
11     associate  action+ {
12       o.update(s);
13     }
14   }
15
16   public void associate(Subject s, Observer o){}
17   public void release(Subject s, Observer o){}
18 }
```

Figure 4: Tracematch implementing the Observer pattern, translated from the relational aspect in Figure 3

The definition of the abstract pointcut remains untouched, as do all non-relational members of the aspect (line 2). Then, the following steps are executed for each relational aspect.

1. The aspect parameters are removed, as is the **relational** modifier.
2. All original definitions of relational advice are removed, as now equivalent tracematches reside in the aspect.
3. Last but not least, the associate and release methods are added.

Note that the body of those methods in 3. can be empty. The methods are just required to provide the programmer with a name that she can call and which the symbols can match on.

Observe how similar this tracematch implementation is to the one we showed earlier in Figure 2. In fact, it is almost exactly the same. The only differences are that in Figure 2 we did not take into account de-association via calls to release, and that in the case of the relational aspect, the observer registers itself with a subject by a call to the appropriate aspect, not to the subject directly. This strong correspondence demonstrates that each relational aspect has a *natural* counterpart in the world of tracematches.

To be clear, we wish to point out that it is *not* our intent to generate those tracematches and then present them to the user (who then would have to weave them in turn). We rather implemented this transformation directly inside the AspectBench Compiler, so that it is hidden from the user. The programmer hence does not need to know anything about tracematches to use relational aspects.

4.2 The issue of storing state per association

The translation we gave so far is very straightforward and shows a beautiful, complete correspondence between relational aspects and tracematches. However, there is one language feature, which we consider as crucial, that has not yet at all been handled: The possibility of storing state per association.

In our introduction of association aspects we pointed out that these allow to store values per association. In Figure 1 a time stamp was stored, remembering the last time when a *specific* observer was notified about an update to a *specific* subject. In our operational semantics we defined that relational aspects should be able to use the same feature as well. Every instance field needs a distinct copy per association. This is not yet satisfied by our translation. So far, we left all non-relational members of the aspect untouched. Since the resulting aspect is a singleton, there will be exactly one copy of each instance field.

To correct this, we need to make sure that (1.) we can create aspect instances on-the-fly, (2.) the correct aspect instance is associated with each association and (3.) we delegate all accesses to this aspect instance that would otherwise have gone to the "**this**" receiver.

4.2.1 Creation of aspect instances

In order to create an aspect instance per association, we change the previously empty body of the associate (..) method to the following definition.

```
public static SimpleObserver associate(Subject s, Observer o) {
  return new SimpleObserver();
}
```

Note that the creation of an aspect instance via a constructor call is not actually allowed in AspectJ. Hence, the above code would not compile with a normal AspectJ compiler. However, the implementation of the associate (..) method is never exposed to the user. Instead, this transformation is done purely in our compiler back-end, which is naturally free to generate such code.

The resulting aspect instance can then be captured by the tracematch. The code for the observer tracematch is changed to the following:

```
tracematch(Subject s, Observer o, SimpleObserver so) {
  sym associate after returning(so):
    call(* SimpleObserver.associate (..)) && args(s,o);
  sym associate_again after returning:
    call(* SimpleObserver.associate (..)) && args(s,o);
  sym start before:
    execution(public static void main(String[]));
  ...
  (start | release) action* associate (associate_again* action)+ {
```

We add an additional tracematch parameter so. On association, this parameter is bound to the return value of the associate (..) method—the newly created aspect instance. We only want to capture the aspect instance on the *first* call to associate (..) after program start or after a call to release (..) (on the same values). To do so, we define an auxiliary symbol associate_again that is similar to associate but ignores the returned aspect instance, and a second auxiliary symbol start, that matches the program start. In result, the regular expression

(start | release) action* associate (associate_again* action)+

then leads to the tracematch body being executed whenever action occurs, but only on the *first* aspect instance that was associated with the given variable binding after start or release.

4.2.2 Look-up of the correct aspect instance

The correct aspect instance is looked up automatically, simply by the definition of the tracematch semantics. In the above mentioned code, it would automatically be bound to the variable so.

4.2.3 Delegating to the aspect instance

In order to make the tracematch body access the looked up aspect instance instead of the default "**this**" receiver, we must replace all calls to instance methods and all accesses to instance fields by calls to the aspect instance (in the example, to the object so). On the Java source level, this would be very awkward to do because we would first have to resolve which accesses and calls are made to "**this**"(they do not even have to be prefixed by the qualifier) and then we would have to replace the qualifier accordingly. Instead we opted for an easier way. The AspectBench Compiler uses an internal representation called Jimple. In Jimple, all field accesses and method calls are performed only on local variables. A method call foo() is modelled by a load of the predefined constant @this into a local variable l_this, followed by a call to l_this.foo(). Furthermore, the compiler contains a refactoring that assures that only one such variable exists for the entire method body and that this variable is initialized in the first line of Jimple code. Hence, all we have to do is apply this refactoring and then replace the assignment

```
l_this = @this;
```

by:

```
l_this = so;
```

The value of so is received from the tracematch implementation, more exactly, the *disjunct* that holds the final variable mapping with which the body is to be executed (see [1] and Section 5.2 for details on disjuncts). That way, any method call or field access that previously would have been performed on the "**this**" receiver, is now performed on the *associated* aspect instance.

5 Feature comparison

In this section we comment on the benefits of implementing relational aspects not directly, but rather through a transformation into tracematches. As we show here, the resulting implementation automatically inherits a wealth of features directly from tracematches. Consequently, the implementation is more general than existing ones. Table I gives an overview of those features and in the following sections, we discuss each feature in detail. As the table shows, two features of association aspects are currently not supported by our solution; we comment on those as well.

5.1 Thread safety

Neither association aspects nor relationship aspects are thread safe, as none of them use any synchronization feature. As a consequence, if any association is updated by multiple threads, this might lead to undefined behaviour using either approach.

The implementers of tracematches, however, spent a lot of effort on making their implementation not only thread safe but present a fine-grained locking scheme that allows for a large amount of parallelism. Our implementation of relational aspects inherits this feature. A relational aspect can hence safely and efficiently be updated by multiple threads. As our benchmark section will show, providing thread safety comes at a cost, as there is a non-negligible runtime overhead associated with locking.

5.2 Memory safety via leak elimination

Apart from thread safety, memory safety is also an important issue. What should happen if an object that is associated with some aspect becomes subject to garbage collection? Should the association be released, allowing the object to be discarded? Or should the association be strong in the sense that it keeps the object alive?

We argue that associations should have a *weak* semantics. If an object becomes subject to garbage collection this is because it is not any more strongly reachable by any code in the program. Consequently, in the remainder of the execution no joinpoint could ever be triggered involving the object in question. Hence, there is no point in keeping the object alive, simply because there is no way of ever referring to it again.

In seldom cases where a relational aspect would still like to strongly reference an associated object, it can do so by *manually* storing a strong reference within the aspect. This is much easier than the other way around, where strong references would be the default and the user would then manually have to resort to using the `java.lang.ref.WeakReference` class of the JDK.

Fortunately, because of the way we implemented our relational aspect to tracematch transformation, we get this weak semantics for free. In recent work, Avgustinov et al. [5] proposed an optimization technique called leak elimination. This technique addresses the problem of garbage collecting internal tracematch state

Feature (Section)	Association Aspects	Relationship Aspects	Relational Aspects
implementation approach	compiler	library	compiler/ tracematches
storage of association (5.2)	ITDs	ITDs/ Hash maps	Constraints
thread safety (5.1)	no	no	yes
memory safety (5.2)	yes	no	yes
non-weavable objects (5.3)	no	yes	yes
primitive-value binding (5.4)	no	yes	yes
per-thread association (5.5)	no	no	yes
fast lookup by indexing (5.6)	yes	yes	yes
per-association state (5.7)	yes	yes*	yes
associated(..) pointcut (5.8)	yes	no	no
sharing (5.9)	yes	no	no
n-ary associations (5.10)	yes	no	yes
dynamic aspect enablement (5.11)	no	no	yes

*to be done manually by the programmer

Table I: Features of the three different implementation strategies (ITD = inter-type declaration) through the *automatic* use of weak references. Their leak elimination algorithm performs a static analysis of the tracematch state machine, determining for each state which variables must be bound at this state and which variables must be rebound before reaching a final state from this state. Using this information, weak references are held to objects at all places where it is allowed by the semantics. (Strong references are still sometimes necessary, e.g. if a value is used in the tracematch body and is not guaranteed to be rebound before hitting a final state.) We designed our transformation specifically in such a way that no additional strong references to associated values are created.

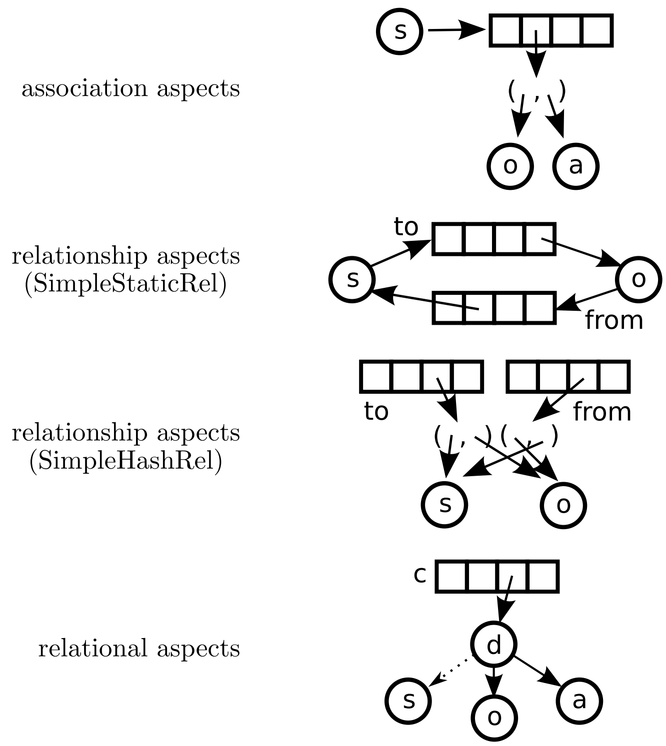

Figure 5: Storage organization for the Observer pattern (s = subject, o = observer, c = constraint, d = disjunct, a = aspect instance); dashed arrow depicts weak reference

Figure 5 shows the storage organization for our subject/observer example, in all three approaches: association aspects, relationship aspects and our implementation of relational aspects. (Relationship aspects provide different means of implementing associations, *SimpleStaticRel* and *SimpleHashRel*.) In order to understand the rationale for this storage structure, let us reconsider the observer advice, here in the syntax of association aspects:

after(Subject s, Observer o) :
  subjectChanged(s) && **associated**(s,o) {...}

Note that the pointcut itself binds the value s. Therefore s does not need to be looked up; it is directly available. However, the implementation *does* need to look up all associated observers o. In association aspects, the compiler hence generates a hash map, which is stored as a field in the type Subject via an inter-type declaration [15]. This hash map has observers as keys. For each observer, it looks up the associated aspect instance. Note that this implementation is memory-safe. If s ever becomes subject to garbage collection, it can be collected, because no additional references to s are created. When this happens, all (strong) references to o held in the association list are collected as well. The observers o, on the other hand, may not be collected as long as another associated Subject s1 is present. This is because if s1 is updated, observers o ought to be notified. The implementation of association aspects correctly satisfies those constraints.

In relationship aspects, things look a little different. Relationship aspects are implemented via a library.

As opposed to the compiler for association aspects, this library can have no knowledge about the direction of look-up that is needed. If *SimpleStaticRel* is used, it soundly over-approximates, providing look-up facilities in *both* directions. This is convenient, however, it implies that strong references to both s and o exist. As a consequence, *both* have to become subject to garbage collection so that *any* of them can actually be collected. In particular, a subject can only be collected when all associated observers are collected. As our benchmarks show, this can lead to a significant increase in memory usage. Furthermore, this problem is not easily solvable with a library approach. The *SimpleHashRel* uses strong references, which is obviously not memory-safe. Note that just using a *SimpleHashRel* with weak keys and values would not suffice, as observers ought to be referenced with strong references. A map with weak keys could do the job, but making such a choice demands quite a bit of insight from the side of the programmer.

The storage organization for relational aspects looks again different. The automaton state for the action holds a constraint, which can be seen as a set of so-called disjuncts. Due to the leak elimination analysis, the disjunct class is generated in such a way that each disjunct holds a weak reference to subjects but a strong reference to observers (for the same reasons as noted above). Hence, if a subject becomes subject to garbage collection, it *can* be collected, yielding a disjunct with an empty slot for s. The next time any transition on this state is made, the tracematch implementation will see that a slot has become empty and hence discard the entire disjunct, deleting all strong references to associated observers. As our benchmarks confirm, this process makes the tracematch-based implementation just as memory-safe as the one of association aspects. However, in contrast, the tracematch-based implementation may need two rounds of garbage collection, with an intervening automaton transition, in order to free all possible memory. As [5] showed, the negative effects of this fact are negligible, though.

5.3 Association of objects of non-weavable classes

The storage organization depicted in Figure 5 exposes one serious implication of the way both association aspects and the SimpleStaticRel of relationship aspects organize their storage of associations. Both implementations introduce fields into s and o. But what if the types Subject or Observer are not weavable? Usually, all types in the Java runtime library are not woven into. This is a frequently recurring issue. In association aspects, it turns out that there is no way of associating objects from such classes. As verified with their developers, if one tries to associate a non-weavable class, e.g. a String value, a `NoSuchFieldError` is thrown at runtime.

Relationship aspects implement the second relationship type, SimpleHashRel, especially for the purpose of associating objects of non-weavable types. This relationship aspect would store associations as mappings from subjects to observers (and the other way around). However, again, this is not memory-safe.

As Figure 5 shows, our implementation of relational aspects does not introduce any new fields onto s or o. Hence, neither the type Subject nor Observer need to be weavable. Objects of any type can be associated with relational aspects.

5.4 Associating primitive values

Because of the same reason, it is no problem to associate a relational aspect with primitive values such as booleans, ints and floats. Very much from the beginning, tracematches [1] already supported the binding of primitive values. The semantics are based on comparison by value, not by reference. In fact, there is no reference. Because the code for all disjuncts in tracematches is generated in a strongly typed way, the generated code uses those primitive types directly. In particular, it does not box the values into objects. This implies that primitive values cannot be garbage collected. Relational aspects directly inherit this useful feature.

Figure 6, for instance, shows the implementation of a cache for values of type List, indexed by values of the primitive type long. A non-relational advice is triggered after a return from a call to factorization (..), which is assumed to take a long time to execute. It captures the return value, a list of long values. This list is associated with the input number by invoking associate(k,v).

If factorization (..) is called with a number already assigned, the relational around advice will apply instead, because the argument then matches the stored key. In this case, we just return the associated return value. Due to the advice ordering and AspectJ precedence rules and because there is no **proceed**() statement, the original joinpoint (the lenghty computation) is not executed and neither is the first piece of advice.

```
relational aspect LongCache(long key, List value) {

  after(long k) returning(List v):
    call(* Factorization.factorize(long)) && args(k) {
      associate(k,v);
  }

  relational Object around(key):
    call(* Factorization.factorize(long)) && args(key) {
      return value;
  }
}
class Factorization {
  static List factorize(long l) {
    /* compute list of factors ... */
    return factors;
  }
}
```

Figure 6: Relational aspect for caching of **long** values

Because no fields can be introduced to primitive values, neither association aspects nor the SimpleStaticRel of relationship aspects can bind primitive values. The SimpleHashRel however, is perfectly suited for this purpose.

5.5 Per-thread and global association

By default, tracematches are instantiated globally. They can also be instantiated per-thread using the **perthread** modifier. If this is the case, they only execute if the observed events executed on one and the same thread. This way, each execution gets its own thread-local scope, which might be useful for some relational aspects.

Neither association aspects nor relationship aspects support per-thread state directly as a language feature.

5.6 Fast look-up through optimized indexing

In more recent work Avgustinov et al. proposed [4, 5], two optimization techniques for tracematches, implementing an enhanced code generation. The first of those techniques is called indexing. It addresses the issue of fast access to the stored tracematch state. Depending on which symbols are most likely to occur on the execution trace, it might be more beneficial to index on certain tracematch variables than on others.

Some other implementations of runtime monitoring [9] use multiple (i.e. all possible) indexing structures to look up variable values, similar to the relationship aspects library. However, this naturally increases the memory footprint of the running program. In [5], the authors propose a heuristic that selects variables for indexing automatically. However, since it is a heuristic, it does not always yield optimal results. Yet, the algorithm can be given a clue in the form of an annotation, with the keyword **frequent**, as to which symbols are believed to occur frequently on the execution trace.

Luckily, for tracematches implementing relational aspects, the place where such an annotation should go is very clear. The action symbol will, in virtually all cases, be much more likely to match than the symbols associate and release. Hence, we simply add the following line to the tracematch definition, giving the clue that actions occur more frequently than other symbols:

frequent action;

Association aspects also choose their indexing structure based on the look-up direction. Consequently, look-up is guaranteed to be fast. A field load to retrieve the hash map, followed by a hash map look-up is all that is needed to look up the correct aspect instance.

Relationship aspects provide equally fast look-up, by similar means. The only difference is that look-up data structures are kept in memory for both look-up directions. Although there is never any look-up from observers to subjects, this association is still stored. As our benchmarks show, this leads to increased memory usage.

5.7 Per-association state

In Figure 5 we can clearly see that association aspects as well as our relational aspects associate a unique aspect instance with each single association. This allows for storage of per-association state. Through the indexing structures, look-up of such state virtually comes for free in terms of runtime.

Relationship aspects support per-association state as well, but in a manner which requires some effort from the programmer. Some relationships may be given a third parameter, a class which essentially holds the state of a given association.

5.8 Symmetric look-up

Association aspects allow for a unique feature, the **associated**-pointcut. This pointcut allows for symmetric look-up of associated objects. If a pointcut

target(x) && (**associated**(x,y) || **associated**(y,x))

is attached to an advice, this advice is executed multiple times, for all cases where x is associated on the right-hand side or left-hand side of the association aspect.

This feature is currently *not* supported by our implementation of relational aspects (nor by the relationship aspects library). However, as Pearce and Noble showed [16], symmetric relationships can simply be programmed by automatically associating a tuple (y,x) via an advice, whenever associate (x,y) is called by the programmer. While this comes at a cost of using additional memory for storage, it retains the functionality of symmetric look-up. We expose such an implementation in the download package for our compiler.

5.9 Sharing

As Sakurai et al. note in [19], association aspects use sharing for look-up tables: If there are two uses of the **associated**(..) pointcut which access the same parameters at the same positions, one single look-up suffices for the evaluation of both pointcuts. Our tracematch-based relational aspects unfortunately do not support such sharing yet. If the same relational aspect contains n pieces of advice, on a call to associate, association will happen n times. Further, if different pieces of relational advice share the same joinpoints as actions, at such a joinpoint, the related aspect instance is looked up multiple times, one time for each match.

We believe that sharing would in fact be very appealing. Indeed, we thought about sharing before, on the general level of tracematches. Tracematch definitions that share common joinpoints could be evaluated in common by merging their finite state machines. As so often, the devil is in the details and such sharing would largely complicate the tracematch code generation. Hence, we leave this feature to future work.

5.10 n-ary associations

The relationship aspects library does not support general n-ary relations for $n \neq 2$. This is likely due to the fact that one would have to implement at least *o(n)* different generic aspects in their library in order to allow up to n-ary relations. Since the implementations of both association aspects and relational aspects are based on code generation, such scalability issues do not exist. The appropriate data structures are generated for any $n > 0$.

5.11 Dynamically enabled aspects via nullary associations

A special case is the nullary association. At AOSD 2005, there was a "Birds of a Feather" session on per-instance aspects, where the issue was raised that at the very least, AspectJ should have a means of enabling or disabling aspects at runtime[3]. Right now, AspectJ does not support dynamic enablement of advice. This shortcoming is frequently being worked around by guarding all pointcuts of pieces of advice that should be dynamically enabled with a prefix "if (b) &&" where b is a static boolean field.

Relational aspects allow for dynamic disablement by associating/releasing the empty object vector of length 0. By declaring a relational aspect with an empty parameter list, one gets an aspect in which all relational advice are disabled by default. After a call to associate (), all those pieces of advice are enabled, a call to release () disables them again. In this case, instance fields of the aspect automatically exist exactly once, as is usually the case for AspectJ aspects that are declared as **singleton** (the default in AspectJ).

Association aspects do not allow for the association of an empty vector. They *cannot* do so because associations are stored on objects. If there is a nullary association, which object should the association be stored on? In relational aspects, the association is stored in the disjunct, as implemented by the standard operational semantics for tracematches [1]. Since the relationship aspects library only allows for binary relations, it also has no support for dynamically enabled aspects.

This concludes our feature comparisons of relational aspects with previous approaches. As we saw, many synergistic effects arise from implementing relational aspects via tracematches, yielding a plethora of useful features and immense flexibility. As we show now, we can even define a new language feature, a *relational tracematch*, that combines the possibility of explicit object association with the usual benefits of trace matching.

6 Relational tracematches

We wish to motivate relational tracematches by an example. Assume we have a caching concern, similar to the one addressed in Figure 6. The cache in that figure is very basic. Every key/value pair that is cached until the program shuts down. However, in many applications a cache might have to be invalidated, e.g. because the cached computation depends on some globally accessible value that recently changed.

This situation can be expressed as a tracematch pattern. We want to return a cached object if (1) it has been cached before, (2) it is about to be computed/created again and (3) in between, the cache has not been invalidated. Figure 7 shows a relational tracematch that makes use of this observation. It caches String creation via the flyweight pattern [11]. For the sake of simplicity we here assume that the String constructor takes a single argument that uniquely defines the String's content.

Line 1 holds the header of a relational aspect declaring that it associates an object (the parameter) with a String value. The non-relational advice in lines 3-6 implements the association necessary for the cache: Whenever a String is created, this String is associated with the parameter that was passed into the constructor. As mentioned before, the programmer should be able to invalidate the cache. Hence, we provide a method stub invalidate () in line 8. Lines 10-16 finally hold the actual relational tracematch. Because its last symbol is an around-symbol, it declares a return type —String— in line 10. Note that, also in line 10, it declares *no* input parameters. This is because the relational aspect parameters key and value are already

[3] See Adrian Colyer's blog at `http://www.aspectprogrammer.org/blogs/adrian/2005/03/perinstance_asp.html` for more details.

```
1  relational aspect Cache(Object key, String value) {
2
3    after(Object k) returning(String v):
4      call(String.new(..)) && args(k) {
5      associate(k,v);
6    }
7
8    static void invalidate() {}
9
10   relational String tracematch() {
11     sym invalidate before: call(void Cache.invalidate());
12     sym create around(key): call(String.new(..)) && args(key);
13     create {
14       return value;
15     }
16   }
17 }
```

Figure 7: Aspect with relational tracematch caching String creation, allowing for invalidation

visible in the tracematch and no other values need to be accessed. Line 11 declares the symbol invalidate matching calls to the respective method. Line 12 declares the symbol create matching the actual String creation. This symbol will only be matched if the argument at that joinpoint was already associated as key. The tracematch body is defined in lines 13-15. It simply states that when a create occurs (on associated values!) we return the appropriate value.

6.1 Semantics of relational tracematches

The semantics of relational tracematches naturally follow from the ones of tracematches and relational advice. A relational tracematch executes whenever its non-relational counterpart executes, but *only* if all bound values have actually previously been associated.

While a non-relational tracematch is evaluated over each suffix of the entire execution trace, a relational aspect, associated with an object vector v, is evaluated on the sub-trace starting at the first call to associate(v), and ending at the first call to release(v) thereafter.

6.2 Implementation

The implementation of relational tracematches is a generalization of the one of relational advice. A relational tracematch is reduced to a non-relational one by the following steps:

- Add to the tracematch parameters the parameters of the declaring relational advice. Further, add the auxiliary parameter for holding per-association state.
- Add symbols associate, associate_again, release and start in the same way as for relational advice.
- For each **around**-symbol s, add a **before**-symbol named s_before with the same pointcut.
- If r is the original regular expression of the relational tracematch, replace r as follows.
  1. Let rs be the *shuffle* of r and associate_again∗, i.e. the copy of r where any primitive symbol s in r was replaced by "associate_again∗ s".
  2. Let rs_na be the copy of rs where every occurrence of an **around**-symbol s was replaced by s_before (see above).
  3. Let syms_na be the disjunction of all symbols of the non-relational tracematch's pattern, again with **around**-symbols s replaced by s_before.

4. Let skip be the disjunction of all other declared symbols of the non-relational tracematch.
5. Then finally replace r by the following regular expression:
   (start | release | skip) syms_na∗ associate (rs_na)∗ rs

- Transform the tracematch body to refer to the auxiliary state variable instead of "**this**", as before for relational advice.

Step 1 takes care of properly ignoring redundant associations of already associated values. Step 2 establishes a necessary invariant for **around**-symbols (see [1]): An **around**-symbol must only occur in the final position of a regular expression. Step 4 is necessary to allow spurious events between start or reset and the first association thereafter. Figure 8 shows the non-relational aspect induced by the relational aspect in Figure 7.

Because of the tracematch's scope, a call to invalidate () indeed invalidates the cache in our example. For instance, assume that a program calls this method after an association, and then triggers "create". This would give us the following trace.
{start}
{associate , associate_again}
{ invalidate}
{create , create_before}

Note that the regular expression does not match any suffix of this trace, with no variable binding. Now assume that the program performs another association, followed by another "create" event. This leads the new trace:
{start}
{associate , associate_again}
{ invalidate}
{create , create_before}
{associate , associate_again}
{create , create_before}

Note now that the regular expression matches the partial suffix trace
invalidate create_before associate create .

6.3 Relational advice are special relational tracematches

It is interesting to note that in the same way as an advice is a special case of a (very simple) tracematch, a *relational* advice is a special case of a *relational* tracematch. Indeed, our compiler extension implements relational advice not quite as previously stated in Section 4 but rather by first converting the relational advice into an equivalent relational tracematch that has only one symbol, action, and a regular expression of the form "action". This relational tracematch is then converted using the above mentioned procedure.

7 Performance Evaluation

After we realized how much flexibility we could gain by implementing relational aspects via tracematches, we were naturally interested in the question at which cost this level of flexibility would come. As we saw in Section 5, most flexibility comes from the unique storage organization that is intrinsic to tracematches. However, this storage organization uses more indirections than the ones of the other two existing approaches. Therefore we would assume an increased runtime cost.

We conducted the following experiment to determine the runtime cost and memory efficiency that is induced by each of the three implementations, association aspects, the relationship aspect library and relational aspects using tracematches. Because of the different limitations of the various approaches depicted earlier in Figure I, we had to choose a simple example aspect that can be implemented with all three approaches. In [18, 19], Sakurai et al. use an *Equality* relation (originally pointed out by [20] as a concern for systems integration) that keeps two Bit objects equal by associating them with a special instance of the

```
aspect Cache {

  after(Object k) returning(String v):
    call(String.new(..)) && args(k) {
    associate(k,v);
  }

  static void invalidate() {}

  String tracematch(Object key, String value) {
    sym associate after:
      call(* SimpleObserver.associate(..)) && args(s,o);
    sym associate_again after returning:
      call(* SimpleObserver.associate(..)) && args(s,o);
    sym start before:
      execution(public static void main(String[]));
    sym release after:
      call(* SimpleObserver.release(..)) && args(s,o);
    sym invalidate before:
      call(void Cache.invalidate());
    sym create around(key):
      call(String.new(..)) && args(key);
    sym create_before before():
      call(String.new(..)) && args(key);
    (start | release | invalidate) create_before*
    associate (associate_again* create_before)*
    associate_again* create {
      return value;
    }
  }

  /* definition of methods associate/release omitted */
}
```

Figure 8: Non-relational aspect induced by relational aspect with relational tracematch from Figure 7 (auxiliary state variable and frequent-annotation omitted)

aforementioned Observer aspect. A set(), respectively clear() operation is invoked on the one bit whenever the other one is set/cleared. Although this is an easy aspect which can be implemented in association aspects and our relational aspects, it cannot easily be implemented using the relationship aspects library because it uses per-association state. A Boolean flag is set whenever a particular association was updated, to break an otherwise possibly infinite recursion. Although this could be manually worked around with the relationship aspects library, we thought that this would have been an unfair comparison. Hence, we opted for an easier aspect that only propagates equality from the left to the right, having only the right associated bit act as an observer of the left associated bit. Figure 9 shows the relational aspect implementing this functionality.

```
relational aspect Equality(Bit b1, Bit b2) {
  relational after(): call(public void Bit.set()) && target(b1) {
    b2.set ();
  }
  relational after(): call(public void Bit.clr()) && target(b1) {
    b2.clr ();
  }
}
```

Figure 9: Relational aspect implementing “directed equality”

Because of its simplicity, this benchmark might seem not representative for large programs. However, we wish to note that this benchmark excessively exercises the dispatch of relational advice, which we consider the main functionality of relational aspects, association aspects and the relationship aspects library. If one took a larger program as a benchmark, the relative overhead of this dispatch would certainly be smaller, not larger.

Our benchmark driver class first tests the correct functionality of the advice implementation by associating three different bits with the aspect and then updating and checking their values. It then executes 100,000 warm-up rounds. In each round, each bit is set and then cleared again (and the aspect propagates those changes to the associated bits). We then execute the same loop 30 times, which gives us 30 different timing values.

To measure the memory consumption, we then associate 10,000 auxiliary bits with the aspect, on its left-hand side. Those bits actually only need weak references. Hence, they should usually *not* lead to increased memory consumption. We then execute the previous loop another 30 times.

Experiments were performed on an AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ with 4GB RAM. For execution we used the Java HotSpot(TM) 64-Bit Server VM (build 1.6.0-rc-b104) in mixed mode and with standard heap size.

As mentioned in Section 5.1, our relational aspects are the only thread-safe approach because it is the only one that uses locking. This locking comes at a cost. To measure the amount of runtime overhead caused by our locking scheme, we ran our implementation twice, one time with a special version of our runtime library that uses no locking, and one time with our normal runtime library. For the relationship aspect library we used the SimpleStaticRel. In [16] it was shown that it is generally faster than SimpleHashRel.

7.1 Runtime overheads

Figure 10 shows the running times of the entire benchmark. We averaged over the last 20 of each 30 rounds. The error bars show the 95% confidence intervals. The Figure shows four groups of two bars. Each two bars reflect the measurement without and with the 10,000 auxiliary bit objects present. As we can see, association aspects are fastest with the relationship aspects library being slightly slower. Our own tracematch-based implementation is relatively far off. Without locking it is almost 10 times slower than association aspects, with locking about 14 times. As we can see, locking is an important factor, however larger runtime also arises without locking.

We did some profiling to find out why this is so. We found that about 25% of all our runtime overhead is spent in calls to Reference.get(), which is due to our uses of weak references. However, as we showed in

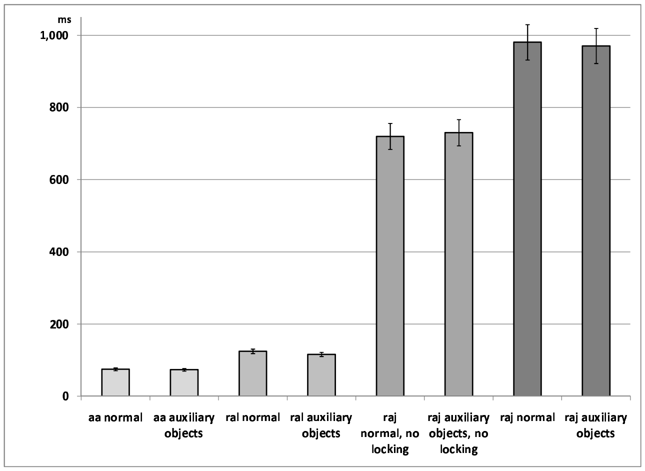

Figure 10: Running times in milliseconds for association aspects (aa), relationship aspect library (ral) and relational aspects (raj, with and without locking)

Section 5.3, the use of such weak references is the only way to implement a memory-safe storage model for objects of non-weavable classes. We conclude that at least this amount of overhead is the necessary cost one has to pay for an approach that offers such a degree of flexibility. The rest of the overhead is due to the more general and hence more complicated storage structures tracematches use. After all, tracematches were not designed with relational aspects in mind.

Despite the fact that our approach executes around 10 times slower than association aspects, it still executes very fast. Note that 100,000 rounds of six relational advice executions each all execute in under one second! This means that even with locking enabled the cost of one single relational advice dispatch and execution is only slightly above 16 microseconds. We believe that any overhead in this order of magnitude is negligible for a programming language feature residing on such a high level of abstraction.

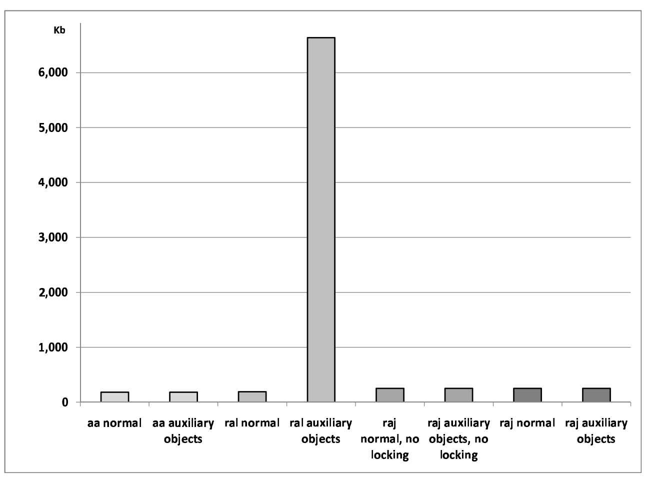

Figure 11: Memory consumption in Kilobytes for association aspects (aa), relationship aspect library (ral) and relational aspects (raj, with and without locking)

7.2 Memory consumption

Figure 11 shows the maximal memory consumption for the same eight runs. Association aspects use about 184Kb. The relationship aspects library uses about 192Kb *without* the auxiliary 10,000 bits present. This slightly higher overhead is caused by the bidirectional storage organization as it was shown in Figure 5. Our own implementation using tracematches uses again slightly more memory, around 250Kb in total. The increased usage is here due to the fact that the tracematch state machine has to store disjuncts.

The only real reason to worry is however the fourth bar, showing the overhead for the relationship aspects library with auxiliary bits present. As anticipated, the implementation is not memory-safe. Although no external strong references to the auxiliary bit objects exist, those objects cannot be garbage collected, neither can their association. This quickly fills up memory.

Discussion We conclude that although relational aspects are slower than existing approaches they seem fast enough. The implementation proves memory-safe.

A full implementation of our approach is available at

`http://www.aspectbench.org/`

along with all raw data, test cases and benchmarks that we used.

8 Conclusions

In this work we presented relational aspects, a new AspectJ language extension. Their semantics are very similar to related work on association aspects. However, the implementation *we* present is based on a reduction to tracematches, another AspectJ language extension, designed for matching on a program's execution history.

As we showed, this implementation scheme yields several benefits over existing implementations. It is the only one that combines important features of thread safety, memory safety, per-association state and binding of primitive values or values of non-weavable classes. Furthermore, our implementation yields a new high-level language feature, relational tracematches. On the other hand, one feature present only in association aspects, sharing of look-up structures, was identified as a useful future optimization for tracematches and our implementation of relational aspects.

Several benchmarks allowed us to compare previous approaches by other researchers with each other and with our own one. Profiling allowed us to give a detailed account about the reasons for relative slowdowns and increases in memory use. The results showed that, quite naturally, the increased flexibility does come at some runtime cost. Yet, we conclude that the resulting implementation is efficient enough for production use.

We believe that our implementation provides a solid foundation for future research in the field, by ourselves and others. In particular, we are interested in a large-scale case study for future work.

Acknowledgements We thank Kouhei Sakurai and his colleagues for fruitful discussions and for clarifications they provided about association aspects. Further we express our gratitude to David J. Pearce and James Noble for making their implementation of relationship aspects available. Equally, we thank the rest of the abc group for making their implementation of tracematches available. Julian Tibble from Oxford University provided helpful comments on how to structure the regular expressions generated for relational tracematches.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Int.*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 345–364. ACM Press, 2005.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of caesarj. *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *Int. Conference on Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

[4] P. Avgustinov, J. Tibble, E. Bodden, O. Lhoták, L. Hendren, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, `http://www.aspectbench.org/`, March 2006.

[5] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Int. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM press, 2007. To appear.

[6] S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In Ernst [10], pages 323–346.

[7] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[8] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In Ernst [10], pages 525–549.

[9] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Int. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM press, 2007. To appear.

[10] E. Ernst, editor. *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*. Springer, 2007.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[12] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Third Edition: The Java Series*. Prentice Hall, 2005.

[14] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Int. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173. ACM press, 2002.

[15] R. Laddad. *AspectJ in Action*. Manning, 2003.

[16] D. J. Pearce and J. Noble. Relationship aspects. In *Int. Conference on Aspect-Oriented Software Development (AOSD)*, pages 75–86. ACM Press, 2006.

[17] D. J. Pearce and J. Noble. Relationship aspects. In *European Conference on Pattern Languages of Programs (EuroPLOP)*, pages 531–546. Universitätsverlag Konstanz, 2006.

[18] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In K. Lieberherr, editor, *Int. Conference on Aspect-Oriented Software Development (AOSD)*, pages 16–25. ACM press, 2004.

[19] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Design and implementation of an aspect instantiation mechanism. *Transactions on Aspect-Oriented Software Development I*, 3880:259–292, 2006.

[20] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages: integration as a crosscutting concern for AspectJ. In *Int. Conference on Aspect-Oriented Software Development (AOSD)*, pages 19–26. ACM Press, 2002.

[21] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[22] K. G. van den Berg and J. M. Conejero. A conceptual formalization of crosscutting in AOSD. In *Desarrollo de Software Orientado a Aspectos (DSOA2005), Granada, Spain*, volume 24/05, pages 46–52. Universidad de Extremadura, September 2005.

[23] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In Ernst [10], pages 54–78.