A Reflexive Extension to Arachne’s Aspect Language

Nicolas Lorian	α  Marc Ségura-Devillechaïseα  Thomas Fritzγ  Jean-Marc Menaudα
α OBASCO research group, EMN/INRIA - LINA, Nantes, France
γ Software Practices Lab, University of British Columbia, Vancouver, Canada
nloriant@emn.fr, msegura@emn.fr, fritz@cs.ubc.ca, jmenaud@emn.fr

ABSTRACT
Aspect weaving at run time has proven to be an effective way of implementing software evolution. Nevertheless, it is often hard to achieve adequate modularization and reusability in face of run time and implementation issues.

Arachne is an AO system that features a run time aspect weaver for C applications, and a language close to the C syntax. In this paper we present a reflexive extension of Arachne’s aspect language. We show through extracts of a deadlock detection aspect, how this extension improves the modularization of crosscutting concerns and the reusability of aspects.

1. INTRODUCTION
Run time weaving has been used to tackle many issues arising after software deployment, such as debugging, profiling, security and software evolution [5, 6].

Nevertheless, run time weaving poses challenges to developers, such as dealing with unknown program state at weave time. For example, in previous work [3] we have proposed an aspect preventing buffer overflow that is only effective for buffer allocated after the aspect weaving because the size of the buffer is only known at allocation time. The point is that the state of the aspect does not correspond to the program state. While this limitation is acceptable in most situations, this issue alleviates the need to inspect and modify aspect state. Reusing aspect code is also difficult, indeed two implementations may slightly differ. Thus, adapting one aspect for another base program often results in complex code. Generating arbitrary events would allow developers to write aspects triggering arbitrary events that would match with another aspect. Hence this would act as an interposition layer between one aspect and another base program.

We believe those failures of AO languages to provide adequate modularization and reusability can be addressed with a reflexive aspect language. To illustrate our proposal, we present in this paper a case study on a deadlock detection aspect. We have built a reflexive extension of Arachne’s aspect language and demonstrate through incremental code snippets how this extension addresses runtime issues while improving aspect modularization and reusability.

The rest of this paper is organized as follows. Section 2 motivates the need for deadlock detection in concurrent programming. Then in Section 3 we emphasize benefits of our reflexive extension through four incremental aspects for deadlock detection. Section 4 discusses the implementation in the Arachne system before we finally conclude.

2. MOTIVATING EXAMPLE
Deadlock is a well-known problem in concurrent programming. For performance reasons, real-world concurrent applications are widely programmed in the C language. Concurrent libraries available for use with C provide the basic synchronization primitives, but give little support for the safety of concurrent programs. For example, the mutex operations provided by the widely used POSIX thread library only provide two checks on mutex usage: (i) no thread will try to acquire the mutex when it has already exclusive access granted on it, and (ii) only the thread owning a mutex may release it. These two properties are insufficient to guarantee a deadlock free program. Static analyzers exist to detect and thus prevent deadlocks, but these tools do not scale to large programs [2]. Thus, reusing multi-threaded libraries is still considered a delicate task.

To address the deadlock problem, run time detection has been proposed. One approach is based on an inactivity threshold [4]. Once a thread has been inactive for a given duration, a potential deadlock is detected. This solution has a low overhead but tends to report many false positives. In addition, even when this approach detects a real deadlock, it is difficult to figure out what minimal set of threads should be killed in order to remove the deadlock. Another approach is to maintain a representation of the sharing of resources between threads (and/or processes) using a resource allocation graph, where resources and threads are represented as nodes in a directed graph. Once a thread requires an exclusive access to a resource, an edge is added from the thread to the resource. Once the exclusive access is granted, the edge is inverted. Threads and resources causing a deadlock form a cycle in the resource allocation graph. This approach is accurate, but suffers from two major drawbacks. First, it requires to identify and monitor any access to a shared resource in the application. Secondly cycle detection algorithms typically run in $O(n + m)$ [1], making the deadlock detection scheme costly.

A solution to the performance problem that still maintains accuracy is to activate/deactivate the detection scheme depending on the application’s performance requirement of the moment. Nevertheless, providing this capability is not trivial, as it requires to take into account lock operations that occurred while the deadlock detection was deactivated.

lufs (Linux Userspace File System) is a Linux kernel module and a userspace daemon that exports the Virtual File System interface to userspace applications. It allows to mount userspace file systems and currently supports numerous file systems such as remote file systems over ssh or ftp.
global variable graph. It would be better to relieve the developer of maintaining the relationship between threads and resources in the graph. Making this protocol explicit is possible with the sequence construct of the Arachne aspect language. The sequence construct of Arachne describes interleaved matching of sequence instances. We call a sequence instance, one matching of a sequence that is in progress. In Arachne, multiple sequence instances may exist simultaneously without necessarily progressing synchronously. In Listing 2, each time the base program issues a call to create_lock (line 11), a new sequence instance starts. Then, the sequence instance waits for one or more calls to require or release (lines 14 and 23) as denoted by the star token on line 28. Finally a call to destroy_lock terminates the corresponding sequence instance. In contrast to the version from Listing 1, the current sequence state tells if the lock is created, destroyed, required, acquired or released. Hence, the developer does not have to save the current state of lock. Thus using a stateful construct may simplify the aspect.

### 3.3 Managing the aspect state

The sequence version expresses the deadlock detection in a single aspect. Nevertheless it is still not fully satisfactory: the previous version still uses a global variable graph to store the resource allocation graph. This is problematic because the variable graph hides the dependencies between pointcuts inside the sequence, making it harder to understand. This is because of thread IDs. Indeed, the sequence only tells what locks are acquired, affected and freed but not by what thread. This is because thread IDs do not appear in the sequence’s pointcuts: thread IDs are not a parameter of operations on locks, but are simply part of the execution state. Thus the relationships between thread IDs, lock IDs, and lock status are not implicitly maintained by the sequence. To tackle this issue, we introduced a reflexive construct in the Arachne aspect language: bind. Listing 3 shows how the bind construct avoids the use of data external to the se-

```c
extern int isDeadlocked(graph_t);
extern void created(graph_t, lock_id);
extern void destroyed(graph_t, lock_id);
extern void required(graph_t, tid_t, lock_id);
extern void acquired(graph_t, tid_t, lock_id);
extern void released(graph_t, tid_t, lock_id);

graph_t graph;

// Sequence implementation
Listing 2: Sequence implementation
seq(
    call(lock_id create_lock() && return(lid)) then {
        created(graph, lid);
    } *
)

Listing 1: Naive implementation
```
extern int isDeadlocked(graph_t);

seq 
{

call(lock_id create_lock() && return(lid1));
{
(before call(void require(lock_id) && args(lid2)
&& bind(tid_t tid, gettid()) && if (lid1==lid2))
then isDeadlocked() ? alarm() : 0;
}

after call(void require(lock_id) && args(lid3)
&& if (tid == gettid() && lid1==lid3))
then isDeadlocked() ? alarm() : 0;
}

||

call(void release(lock_id) && args(lid4)
&& if (tid == gettid() && lid1==lid4));
}

call(void destroy_lock(lock_id) && args(lid4)
&& if (lid1==lid4));
}

Listing 3: The bind construct

sequence. The bind construct modifies the way the sequence handles internal information. It associates to the pointcut it is part of, a key/value pair. In Listing 3 the bind construct (line 7) associates the current thread ID with the lock that is required. Thus, the sequence automatically saves what lock has been acquired by whom. Then, the value can be accessed later on lines 11 and 16. The resulting code is smaller, moreover, the developer no longer deals with maintaining the resource allocation graph, indeed the variable graph is no longer used, the relationships between threads and locks are automatically maintained in the sequence internal data. Thus only the deadlock detection appears in aspects’ advice. Nevertheless, the developer has to modify the function isDeadlocked so that it no longer access the variable graph. Instead, through introspection mechanisms, the developer access the sequence internal state that represents the relationship between locks, thread, and locks’ states.

3.4 Making it reusable

Previous versions of the sequence detect deadlock between threads where locks are handled through the create_lock, destroy_lock, require and release functions. In real-world applications, synchronization is implemented using various APIs. While interfaces may be different, the concept remains the same: locks are required, affected, released and so on. Hence it may be interesting to write a sequence that reasons on “abstract” pointcuts in a way similar to the AspectJ Pointcut Designator. In Listing 4, we add a second lock API to the one we used so far. This second API provides process locks. Function create_Plock respectively destroys_Plock, creates respectively deletes, a process lock. Require and release of locks work in a two step way, i.e., a call to the function prepare_op defines an operation (require or release a lock), then a call to the function proceed_op launches the operation.

Listing 4 shows how to adapt the example to work with both APIs. First (lines 3 to 16), we define two types to unify the representation of process ids, thread ids, and lock ids. We adapted the sequence aspect to those new definitions (line 18). Pointcuts in the sequence no longer match the real API, but instead refer to an abstract API. The rest of the listing (from line 41 to the end) is an interception layer between the real APIs and the sequence that matches “abstract” events. We introduce a reflexive extension: the fakeEvent construct, that is used in aspect’s advices. The developer constructs an event that is matched against every pointcut. For example, the aspect on line 41 matches creation of thread locks. Then in the advice (line 42), the fakeEvent simulates a call to the a_create_lock that matches the first pointcut of the abstract sequence (line 19). Similarly, the aspect on line 57 matches the two steps pattern of the process lock API. Upon calls to the function proceed_op and depending on the parameter OP of the function prepare_op, the fakeEvent construct simulates a call to a_require or a_release.

The example exposes benefits of the fakeEvent construct in terms of aspect reusability. Indeed, the sequence detecting deadlock (line 18) has suffered few modifications compared to its previous version (Listing 3). Moreover, the resulting code shows a clear separation between the problem of detecting deadlocks and implementation dependent issues. Moreover the fakeEvent offers higher abstraction that eases aspect manipulation instead of directly manipulating internal data of the aspect.

3.5 Coping with run time issues

As stated in the motivation of the example, activating deadlock detection at run time to achieve a good performance also requires putting the sequence aspect into the correct state to take into account former lock operations that occurred while the deadlock detection was deactivated. As it is possible to recover the information about lock operations still in effect, using this information and the fakeEvent construct provides the possibility of setting the sequence aspect into the correct state and thus the ability to activate the deadlock detection whenever necessary.

The same mechanism, i.e., adapting the state of the sequence aspect to correspond to the state of the application, is useful for buffer overflow detection as described in [3]. We noticed that in many applications the developer saves the size of buffers for various reasons, not only bound checking, which can then be recovered. This knowledge can be used in combination with the fakeEvent construct to even detect buffer overflows where the buffer was allocated before the aspect for overflow detection was deployed, which was not possible before.

4. IMPLEMENTATION

The goal of Arachne is to provide an efficient and coherent run time Aspect-Oriented system for legacy C applications. Arachne features an application independent aspect compiler, and consistent weaving strategies that comply with the C language and runtime ABI standards for the x86 architectures [7]. We have introduced a reflexive extension of the Arachne aspect language featuring two new constructs: bind and fakeEvent. These extensions mainly consist of providing the according language constructs to the aspect developer and extending the aspect compiler respectively.

The bind construct modifies the way Arachne’s sequence manages internal data. The Arachne sequence is a stateful construct, i.e., it maintains a state between matching of multiple pointcuts. Each time the first pointcut of a sequence is matched, Arachne allocates a sequence instance, a structure that holds instance specific data. In Listing 2, each sequence instance has a member current_gstep that indicates what is the next pointcut to match, and members lid1, lid2, lid3 and lid4 that are return values of lock operations. Arachne
extern int isDeadlocked();

typedef struct {
    pid_t pid;
    ufid_t uid;
} ufid_t;

afid_t getufid() {
    return (afid_t) {getpid(), gettid();}
}

typedef struct {
    // D = proc lock, else thread lock in proc ‘pid’
    id_t tid;
} ulid_t;

seq(
    call(ulid_t a_create_lock()) && return(ulid1)) ;
    
    ( before call(void a_require(ulid_t)
    && args(ulid2)) && bind(ufid_t uid, getufid())
    && if (ulid1==ulid2))
    then isDeadlocked() ? alarm() : 0;

    after call(void a_require(ulid_t)
    && args(ulid3)) && if (ufid == getufid())
    && if (ulid1==ulid3))
    then isDeadlocked() ? alarm() : 0;

    | call(void a_release(ulid_t)) && args(ulid4)
    && if (ufid == getufid())
    && if (ulid1==ulid4));

    call(void a_destroy_lock(ulid) && args(ulid5)
    && if (ulid1==ulid5));

    call(lock_id create_lock() && return(lid)) then
    Arachne.fakeEvent( "CALL",
    "ulid_t a_create_lock()", (getpid(), lid) );

    call(Plock_id create_Plock() && return(plid)) then
    Arachne.fakeEvent( "CALL",
    "ulid_t a_create_lock()", (0, plid) );

    call(void require(lock_id)) && args(lid)) then
    Arachne.fakeEvent( "CALL",
    "void,a_require(ulid_t)", (getpid(), lid) );

    call(void release(lock_id)) && args(lid)) then
    Arachne.fakeEvent( "CALL",
    "void,a_release(ulid_t)", (getpid(), lid) );

    seq(
    call(void prepare_op(Plock_id, char*))
    && args(plid1, DP));

    call(void proceed_op(Plock_id) && args(plid2)
    && if (plid1==plid2)) then {
        switch (OP) {
        case REQUIRE:
            Arachne.fakeEvent( "CALL",
            "void,a_require(ulid_t)", (0, lid) );
        case RELEASE:
            Arachne.fakeEvent( "CALL",
            "void,a_release(ulid_t)", (0, lid) );
        }
}

    call(lock_id destroy_lock() && return(lid)) then
    Arachne.fakeEvent( "CALL",
    "ulid_t a_destroy_lock()", (getpid(), lid) );

    call(Plock_id destroy_Plock() && return(plid)) then
    Arachne.fakeEvent( "CALL",
    "ulid_t a_destroy_lock()", (0, plid) );

Listing 4: The fakeEvent construct