

CIF: A Framework for Managing Integrity in Aspect-Oriented Composition

Andrew Camilleri, Geoffrey Coulson, Lynne Blair
{a.camilleri,g.coulson,l.blair}@lancaster.ac.uk

Computing Department
Lancaster University
LA1 4WA, UK
<http://www.infolab21.lancs.ac.uk>

Abstract. Aspect Oriented Programming (AOP) is becoming increasingly accepted as an approach to deal with crosscutting concerns in software development. However, AOP is known to raise software integrity issues. For example, join point shadows may easily omit crucial join points or include inappropriate ones. In this paper, we propose an extensible framework called CIF that constrains aspect-oriented software design and composition with the intent to maintain the integrity of the final composed system. CIF controls the composition of aspects and the base application in three dimensions: where the composition occurs, how the composition is carried out and what exactly is being composed. The framework is intended to be used in a team-based software development environment. We demonstrate the applicability of the framework through an application case study.

1 Introduction

Aspect Oriented Programming [1] is becoming increasingly accepted as an approach to deal with crosscutting concerns in software development. However, the flexible approach to modularity and composition enabled by AOP has a potential downside: it can easily result in systems that behave in unintended and unanticipated ways – i.e. the *integrity* [3] of the system can easily become compromised (examples are given in the following paragraphs). According to our analysis, integrity violations can be conveniently grouped under three orthogonal headings: i) *where* an aspect-oriented composition occurs, ii) *how* the composition occurs and iii) *what* exactly is being composed.

‘Where’ refers to the *quantification* [2] of aspect composition as specified by a pointcut expression. Quantification is about choosing the ‘shadows’ [5] within which composition will occur. The difficulty here lies in getting the strength of the quantification right (and therefore the extent of the shadow). Over-strong (or over-specific) quantification can lead to compositions that miss crucial join points, whereas weak quantification can include unintended join points [30]. In either case the integrity of the resulting system can clearly be compromised. It is also worth noting that threats to system integrity posed by inappropriate quan-

tification are commonly tangled with issues of maintainability and reusability – e.g. there is often a strong desire for weak quantification to help foster aspect reuse (i.e. to avoid ‘fragile pointcuts’ [7]).

‘How’ refers to the compositional semantics employed when weaving aspects; this is especially an issue where multiple aspects are woven at a common join point. Existing AOP systems provide considerable flexibility in this area, offering for example, options such as advice ordering; execution options like ‘before’, ‘after’ or ‘around’; compositional options like ‘call’ or ‘execution’; or aspect instantiation semantics such as ‘per-instance’, ‘per-class’, ‘per-call’ or ‘per-thread’. Inadvertent misuse of these dimensions of flexibility can easily result in integrity violations. For example, if a ‘distribution’ aspect is inadvertently woven before an ‘authentication’ aspect at the same join point, then the resultant system could erroneously make unauthenticated calls. As another example, it is desirable to limit the use of ‘around’ advice: reasoning about aspects that use ‘around’ advice is hard because the advice developer has the luxury to forfeit the original join point (i.e. not call `proceed()`).

Finally, ‘what’ refers to the functionality encapsulated in an aspect. Inadvertent inclusion of inappropriate behaviour in an aspect can again lead to integrity violations; for example, deadlock might result if an aspect unexpectedly accesses a resource shared by the base code. It is also important to have confidence that a given aspect behaves as its name would suggest. For example, it could easily lead to confusion, especially in a team-based software development environment, if an aspect called ‘logging’ failed to call `proceed()` or used apparently inappropriate modules such as an authentication library. In general, therefore, the libraries and other code used by an aspect should be closely related to the associated concern. Diverging from this goal at the very least makes modular reasoning considerably harder.

The contribution of this paper is to propose an extensible framework, called CIF (for “Composition Integrity Framework”), that constrains aspect-oriented software design and composition with the intent to avoid threats such as the above and therefore to maximise the integrity of the composed system. CIF is abstract and independent of any tool specific concepts (and as such is generally applicable to a range of AOP environments) and is designed to be employed in team-based software development environments. The constraint mechanisms offered by the framework closely follow the above analysis in terms of ‘where’, ‘how’ and ‘what’.

The rest of the paper is organised as follows. Section 2 introduces CIF’s basic abstractions. Section 3 illustrates the use of these abstractions in an application case study, and Section 4 discusses our current implementation of CIF. Section 5 surveys related work, arguing that while elements of the integrity issue have been addressed in the literature, no work has yet considered the issue sufficiently comprehensively. Finally, Section 6 offers our conclusions.

2 CIF’s Basic Abstractions: Domains, Realms and Configurations

2.1 Overview

The main contribution of this paper is to propose an extensible ‘integrity framework’ called CIF for aspect-oriented software development. Our objective is to specify and enforce limits for aspects in a very general way. Our approach is to specify what is permissible rather than to explicitly restrict behaviour (this is in line with the well-accepted principle of fail-safe defaults [4]). Although our work to date has largely considered compile-time AOP, our CIF framework is inherently agnostic as to compile-time versus dynamic AOP.

In more detail, our approach deals with AOP integrity in terms of programmer-defined, concern-specific, ‘boundaries’ or ‘ringfences’ that we refer to as *domains*. A domain scopes a specific concern and imposes its boundaries in terms of the ‘where’, ‘how’, ‘what’ analysis discussed in Section 1. Each aspect is then allowed to operate only within the limits of one or more associated domains.

On top of domains, CIF provides the higher-level abstraction of *realms*. The simplest function of a realm is to list and organise the set of domains defined for a given system. But beyond that, realms provide a way for domains to be combined and inter-related while preserving domain autonomy. This allows us, for example, to specify which domains depend on each other (like authorization and authentication), or to specify weaving priorities in cases where a join point falls simultaneously within multiple domains. In essence, a realm provides an abstraction over a set of domains and a discipline to preserve the integrity of their composition.

Finally, on top of realms, CIF offers the coarser-grained abstraction of a *configuration* which allows us to use the foregoing abstractions to define multiple alternative instantiations or ‘builds’ of a whole application, each of which may have its own dedicated integrity constraints.

Architecturally, the CIF approach is to avoid interfering with the either the base code of a system or its associated aspects. Instead, CIF domain, realm, and configuration definitions sit *alongside* the system (see Figure 1) which remains oblivious to the existence of CIF. Essentially, CIF separates out the integrity concern (or ‘meta-concern’). CIF also operates throughout the development life cycle. In particular, the domain and realm abstractions constrain the design of aspects; and at build-time, a tool called the CIF verifier (see Section 4) checks the conformance of aspect implementations to domain and realm specifications, while the configuration abstraction (again, with tool support) helps ensure that application builds or deployments maintain pre-specified integrity constraints. CIF can even deal with run-time compositional constructs such as `cflow`.

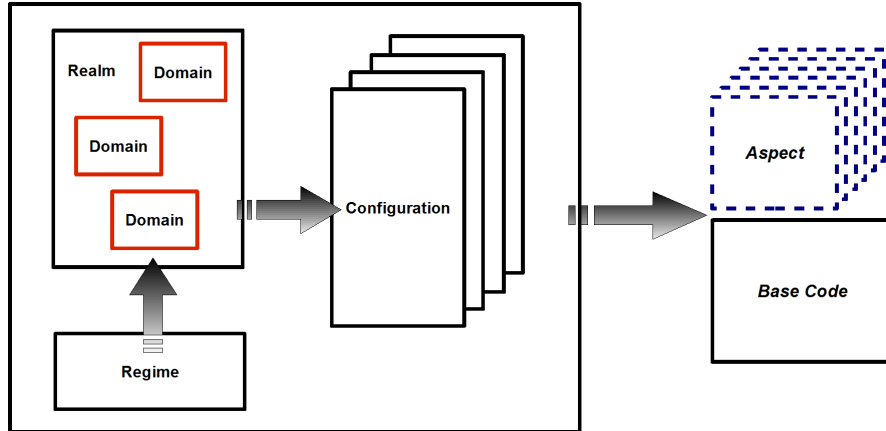


Fig. 1. CIF

2.2 Domains

In this and following sections we use a BNF-like notation¹ to more precisely define the three main CIF abstractions and their supporting concepts. In terms of this notation, a domain is defined in Figure 2.

```

domain ::= <shadow-region, concern-region, permissable-compositional-styles>
shadow-region ::= region
concern-region ::= region
region ::= {programmatic-element-instance}
permissable-compositional-styles ::= <{advice-related-style},
                                     {pointcut-related-style},
                                     {instantiation-related-style},
                                     ...>
advice-related-style ::= BEFORE | AFTER | AROUND ...
pointcut-related-style ::= boolean expression of primitive-pointcut
instantiation-related-style ::= PER-INSTANCE | PER-CLASS | PER-CALL | PER-THREAD ...
primitive-pointcut ::= CALL | EXECUTION | CFLOW ...
programmatic-element-instance ::= this refers to a specific named class, component,
                                method, constructor, etc.

```

Fig. 2. Domain

As can be seen, a *domain* consists of a *shadow-region* (cf. ‘where’), a *concern-region* (cf. ‘what’), and a set of *permissable-compositional-styles* (cf. ‘how’). Before discussing these in detail we first expand on the general notion of a ‘region’. A *region* is a set of *programmatic-element-instances* in the

¹ In this notation < > refers to a tuple; and { } to a set.

target programming language environment such as specific classes, methods, fields, etc. The intention is that a region provides a means to ‘slice’ an application in some appropriate manner using a predicate, which picks up a specific set of *programming-element-instances*. For example, in the case study that will be described in Section 3, we define predicates over the Java package structure of the system: `healthwatcher.view.command.*` in Figure 7 matches all the *programmative-element-instances* (in this case classes) scoped within `healthwatcher.view.command`.

Given this understanding of a region, a domain’s *shadow-region* refers to the set of programmative-element-instances that fall within the scope of this domain. More specifically, when an aspect is assigned to this domain (see Section 2.4), and a pointcut expression is written to specify the composition of this aspect, the shadow defined by that pointcut expression is constrained by CIF to fall within the domain’s shadow region. This is useful in a number of ways; for example, if not so constrained, a pointcut that targets the `execution` join points of a class may inadvertently extend over all of the class’s subclasses which may be implemented by different team members who may be unaware of such a pointcut.

The *concern-region* defines the ‘what’ of the domain; this aims to constrain the behaviour that embodies a particular concern. For example, an aspect implementing a distribution concern can be excluded from making use of the behaviour found in a database library.

Finally, the *permissible-compositional-styles* constituent defines the ‘how’ of the domain. This is expressed as a combination of three other constructs: *advice-related-style* deals with advice types, *pointcut-related-style* is a boolean expression that is written in terms of *primitive-pointcuts*, and *instantiation-related-style* deals with aspect instantiation options. Advice types are useful to constrain (using *advice-related-style*) in a number of scenarios. For example, a synchronization aspect using a semaphore should be restricted to use `before` and `after` for wait and release join points. (It is possible to use an `around` but we do not need to forfeit the original join point. In general, we should always strive for the simpler semantics since it reduces the possibility of introducing errors.) The *pointcut-related-style* is useful because it allows us to define any implicit assumptions which an advice might have. For example in AspectJ, the semantics of the implicit `this`² object varies depending on captured join point [5]. We can easily violate integrity if we change a pointcut (in a manner that changes the context accessed through `thisJoinPoint`) without a corresponding change in how the corresponding `target` object is used. The arguments of an advice can also be a source of confusion and this can be alleviated by an explicit *pointcut-related-style*. We may constrain the exposure of the arguments at a specific join point using the `args` pointcut instead of opting for an advice to access them using reflection. Similar reasoning applies for the *instantiation-related-style*.

The benefits of *permissible-compositional-styles* are especially visible in scenarios where the pointcut is defined separately from the advice implementation;

² The same reasoning applies for the implicit `target` object and the arguments at a join point. In AspectJ these are all accessed through the implicit `thisJoinPoint`.

for example, AspectJ provides the use of abstract aspects. In such scenarios it is possible that implicit assumptions regarding the runtime context are violated (as described above), because each developer may have a different intuition of how composition should occur. This is especially true if we take into account the subtle differences between certain primitive pointcuts; the classic example here is `call` and `execution` (in AspectJ the reference to the implicit `target` object is different for these pointcuts), but another possible example is `within` and `cflow`. These subtle problems are perhaps more common in environments that provide an asymmetric paradigm for the encapsulation of concerns [34, 35].

As shown in Figure 2, the *permissable-compositional-styles* constituent is extensible and can comprise a range of mechanisms, the choice of which is dependent on the target AOP environment in which CIF is being applied³.

To summarise, the primary purpose of a domain is to constrain its associated aspect or aspects. We say that an aspect is suitably constrained by (*conforms to*) a domain iff:

1. all shadows specified by pointcuts applied to the aspect are contained within the domain’s *shadow-region*,
2. all *programmatic-element-instances* used by the aspect’s advice are contained within the domain’s *concern-region*,
3. the aspect’s composition is constrained by its associated domain’s *permissable-compositional-styles* definition.

Examples of the use of the domain abstraction are given in Section 3.

2.3 Realms

The pseudo-BNF for the realm abstraction is shown in Figure 3. As can be seen, a *realm* is a set of one or more domains plus a *regime* that constrains relations between the domains. This regime is a boolean expression that is written in terms of an extensible set of so-called *integrity functions*. The initial set of integrity functions that we have employed in our work to date is shown in Figure 4.

These integrity functions are all of type boolean and can express a variety of constraints between domains. In general, we differentiate between two types of integrity functions: static and dynamic. The key difference is that static integrity functions perform checks that deal more closely to the internal integrity of CIF and thus are applied before the start of the weaving process; dynamic functions are applied during the weaving process, performing checks or possibly influencing it in some manner (more about this in Section 4). In Figure 4 all of the functions are static, except for *precedence()*.

The function *dependent()* declares dependencies between domains: i.e. *d1* is dependent on *d2*. This might be used, for example, to provide a regime for domains that respectively address authorisation and authentication – i.e. the final system can only include authorisation (as a concern) if it also includes

³ Another possible ‘style’ that might be applied in an AspectJ [33] environment, for example, is the `declare` construct.

authentication. Similarly, the function *exclusive()* declares which domains are mutually exclusive i.e. *d1* and *d2* are mutually exclusive. For example, we may design a system that implements distribution as an aspect and want to leave open the possibility of using alternative distribution technologies (e.g. CORBA or Java RMI). In such a case we can construct separate domains for each technology and define them to be mutually exclusive.

The *closure()* function determines the ‘level’ to which the domain’s concern-region is to be verified against an associated aspect. For example, with *level* = 0 only code directly included in the aspect’s advice is verified; with *level* = 1, verification extends to the top level of all the modules and other programmatic-element-instances that are referenced from within the advice; with *level* = ∞ the closure of *all* code referenced directly or indirectly from the advice is verified.

```
realm = <{domain}, regime>
regime = boolean expression involving integrity functions
```

Fig. 3. Realm

```
dependent(Domain d1, Domain d2)
exclusive(Domain d1, Domain d2)
closure(Domain domain, Integer level)
precedence(Domain d1, Domain d2, Region shadow_set)
```

Fig. 4. Example Integrity Functions

Finally, the function *precedence()* imposes a weaving order for advice between domains that share common shadows (thus addressing an element of the ‘how’ dimension that could not be addressed in individual domain definitions). In more detail, *precedence()* defines the advice precedence of *d1* over *d2*; the *shadow_set* argument is used to scope the set of shadows at which the precedence relation should apply⁴.

To round off the discussion of realms, we can summarise by saying that the purpose of a realm is twofold. First, it gives us an abstraction over all the domains that are relevant for a specific application. Second, using regimes we can express constraints over domains with the intent to preserve the integrity of a final application. In general, regimes (and their constituent integrity functions) are useful because they allow us to go beyond the programming language based constraints expressed using domains. Moreover, the fact that new integrity functions

⁴ Incidentally, the precedence relation expressed as a sequence of aspects provided by most AOP environments, can be viewed as a special case of our *precedence()* function.

can be added provides CIF’s main mechanism for extensibility which enables us to cover arbitrary application-specific integrity issues.

An example of the use of the realm abstraction is given in Section 3.

2.4 Configurations

The purpose of our final abstraction, that of a *configuration*, is to enable us to create different application builds from a set of domains. For example, we can use configurations to define application builds that are supported by different OSs, or debugging versions, or versions that include or omit security, etc. Intuitively, a configuration provides us with an instantiation of a realm.

```
configuration = {<domain, aspect>}, realm
```

Fig. 5. Configuration

In more detail, a configuration, as defined in Figure 5, specifies: i) a set of all the domains that are relevant to a particular application build; ii) a congruent set of aspects that are to be associated with and constrained by those domains; and iii) a realm definition that specifies the regime constraints associated with those mappings. Essentially, a configuration embodies the following assertions:

1. that each aspect conforms to its associated domain (or domains – as we have seen, an aspect can be mapped to multiple domains),
2. that the given realm’s regime evaluates to *true*.

An example of the use of the configuration abstraction is given in Section 3.

2.5 CIF Syntax

Figures 2, 3 and 5 defined CIF’s main concepts in an abstract manner. In practice, CIF specifications employ a concrete syntax. Due to lack of space, we do not describe this syntax formally; rather we introduce it by example. More specifically, examples of concrete domain, realm and configuration specifications are given in Figures 7, 8 and 9. These CIF specifications are related to an application case study that will be described in Section 3.

As can be seen in Figures 7, 8 and 9, the various constituents of the framework (regions, styles, etc.) each have their own corresponding labeled sections in a specification, and each of these sections comprises a set of definitions in which CIF-defined attributes are assigned values within curly brackets. Thus, Figure 7 shows a **domains** section which defines a number of domains (in this case two) together with their corresponding attributes – i.e. *shadow* and *concern* regions from the **regions** section, and *permissible-compositional-styles* from the **compositional-styles** section.

The `regions` sub-section is in turn made up of region definitions that are associated with predicate-valued attributes that specify sets of *programmatic-element-instances* that should fall within the region being specified (the `Method` qualifier specifies that methods are the only programmatic element type of interest here). Similarly, the `compositional-styles` section is defined in terms of `advice-related-styles` (with options like `before`, `around`, etc.), `point-cut-related-styles` (with options such as `call`, `execution`, etc.) and `instantiation-related-styles` (with options like `singleton`, `per-thread`, etc.).

Likewise, Figures 8 and 9 respectively define realms and configurations, which employ the same syntactic approach. The `realms` section depends on sub-sections dealing with regimes and integrity functions. The `configurations` section simply lists a set of configurations, each of which is defined as a list of `<domain, aspect>` pairs.

3 Application Case Study

3.1 Case Study Overview

The purpose of the following application case study is to further motivate CIF's abstractions, but also to clarify the way we envisage CIF being used. The application on which the case study is based is discussed in detail in [31, 32]. In outline, the application is called Health Watcher and is a health service complaint logging system which embodies a distributed client server architecture. It was originally implemented in Java, but later refactored to encapsulate cross-cutting concerns using AspectJ. The refactored version employs aspects that address distribution, persistence and transaction concerns. The main reason for choosing this application (besides it being a real system) as our case study is the variety of configurations in which it is available, as shown in Figure 6 (which shows 4 distinct configurations). Thus, for example, the application could be accessed either through the web with a normal browser or using an application installed on the client side (thus explaining the need for different distribution technologies). Moreover, given a specific platform, the application developers had to cater for different options in terms of transaction and persistence mechanisms (in this case relational and object-oriented databases).

3.2 Using CIF's Abstractions

The Health Watcher application provides an ideal ecosystem in which to encounter integrity violations because it is difficult to manage even such a modest set of concerns, given their multiple implementations. It should be obvious that the integrity of the final system can be easily violated if the corresponding aspects are not carefully combined together. CIF is useful in this situation because it gives an abstraction over the concerns and on how they can be combined.

Turning to the specifics of the CIF specification, Figure 7 defines appropriate domains for the application's distribution concern (which we will employ as our

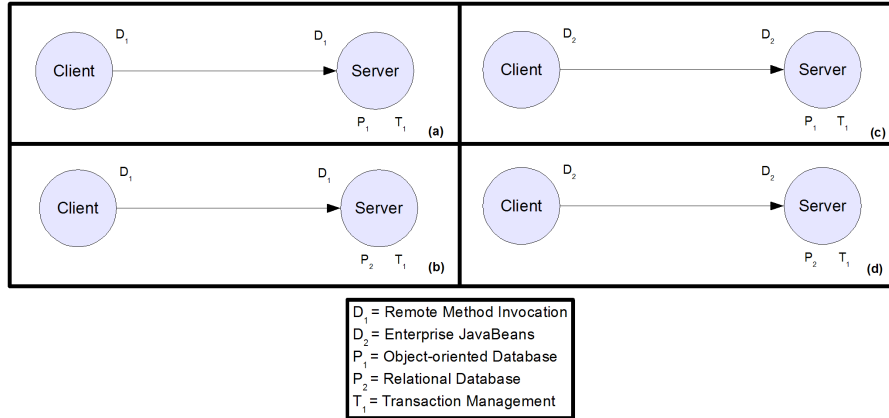


Fig. 6. Health Watcher Configurations

primary example concern). Two domains are specified, one for the client side and one for the server side. The concern-region of `rmiserver-Domain` specifies that only methods from the `java.rmi.server` package can be employed within aspects associated with this domain. This serves to eliminate, for example, an EJB based distribution implementation. The shadow region specified establishes an explicit set of *programmatic-element-instances* that constrain the compositional shadow for this domain. This prevents the aspects associated with this domain from accidentally composing with elements that are not listed in this region.

<pre>[domains] rmiserver-Domain = { shadow-region = server-shadow; concern-region = rmiserver-concern; permissible-comp-style = server-style; } rmiclient-Domain = { shadow-region = client-shadow; concern-region = rmiclient-concern; permissible-comp-style = client-style; }</pre>	<pre>[compositional-styles] server-style = { advice = ad-distribution; pointcut = pc-server; instantiation = inst-distribution; } client-style = { advice = ad-distribution; pointcut = pc-client; instantiation = inst-distribution; }</pre>
<pre>[regions] rmiserver-concern[Method] = { java.rmi.server.*; } rmiclient-concern[Method] = { lib.distribution.rmi.MethodExecutor.invoke(..); java.rmi.Naming.lookup(..); } server-shadow = { healthwatcher.model.*; } client-shadow = { healthwatcher.view.command.*; ... }</pre>	<pre>[pointcut-related-styles] pc-server = { type = execution; } pc-client = { type = call; } [advice-related-styles] ad-distribution = { type = around; } [instantiation-related-styles] inst-distribution = { type = singleton; }</pre>

Fig. 7. Example Domains

The `compositional-styles` section constrains aspect oriented compositions for the client and the server, regardless of the concrete distribution technology. In this case, the `pointcut-related-styles` for the client and server side are constrained to use `call` and `execution`, respectively. This prevents us from confusing the advice arguments (exposed by a pointcut that is defined separately from the advice implementation) for arguments of intercepted methods. If the style for composition is constrained through `compositional-styles` then we can have a uniform compositional view for all distribution concerns and our assumptions will always be sound. For example, we can choose to explicitly expose the intercepted method arguments by constraining the use of the `args` pointcut, instead of relying on each advice to access them through `thisJoinPoint`.

Finally, both domains constrain their `advice-related-styles` to be `around` because the distribution aspects need to marshal data (i.e. method arguments) on both requests and replies. Using a `before` or an `after` here would be an error, since we want to replace the original join point. The respective `instantiation-related-styles` are restricted to be `singleton` since we only want one running instance of the client and of the server. Again, having this kind of constraint capability is useful to maintain a consistency amongst the alternative concern implementations.

```
[realms]
healthWatcherRealm = {
    domains = rmiserver-Domain, rmiclient-Domain,
            ejbserver-Domain, ejbclient-Domain,
            oodb-Domain, relationalDB-Domain,
            transaction-Domain;
    regime = healthWatcherRegime;
}
...

[regimes]
healthWatcherRegime = oodbDependency AND
                    relationalDependency AND serverExclusive AND
                    clientExclusive AND serverClosure;
...

[integrity-functions]
oodbDependency = dependent(transaction-Domain,
                          oodb-Domain);

relationalDependency = dependent(transaction-Domain,
                                relationalDB-Domain);

serverExclusive = exclusive(rmiserver-Domain,
                           ejbserver-Domain);

clientExclusive = exclusive(rmiclient-Domain,
                           ejbclient-Domain);

serverClosure = closure(rmiserver-Domain, *);
...
```

Fig. 8. Example Realms

Figure 8 illustrates realm definitions employed in the case study. As one can see, the regime is a conjunction of five separate predicates. The `dependent()` integrity function is used to state that if a transaction concern is to be employed, a

```

[configurations]
  rmiConfiguration[healthWatcherRealm] = {
    aspect_mapping = (rmIClient-Domain,
      RMIClientDistribution.java);
    aspect_mapping = (rmiServer-Domain,
      RMIServerDistribution.java);
    aspect_mapping = (oodb-Domain,
      OOPersistence.java);
    aspect_mapping = (transaction-Domain,
      OOTransactionManagement.java);
  }

  ejbConfiguration[healthWatcherRealm] = {
    aspect_mapping = (ejbClient-Domain,
      EJBClientDistribution.java);
    aspect_mapping = (ejbServer-Domain,
      EJBServerDistribution.java);
    aspect_mapping = (relationalDB-Domain,
      RelPersistence.java);
    aspect_mapping = (transaction-Domain,
      RelTransactionManagement.java);
  }
  ...

```

Fig. 9. Example Configurations

persistence concern is also required (this is to ensure safe storage for uncommitted data). Similarly, the *exclusive()* function is used to disallow the simultaneous use of RMI and EJB distribution concerns. The *closure()* function specifies that the full closure of the advice code of the server-side distribution aspect(s) should be verified for conformance to the server-side domain's concern-region.

Finally, Figure 9 shows two sample configurations: one employs RMI-based distribution with object-oriented database persistence and transaction management; the other employs EJB distribution, with relational database persistence and transaction management. These two configurations correspond to Figures 6(a) and 6(d). The two configurations specified here correspond to RMI-based configuration and EJB-based configuration in Figure 10 (this figure gives an overview of the whole specification process). It is important to note that the configuration specifications are not effectively specifying any additional integrity constraints. They simply allow us to create our final application. Thus, given a specific realm, the mapping in the configuration can be specified (perhaps using an appropriate GUI) by a deployer who may be ignorant of the details of implementation, but simply requires to build a system which includes a specific set of concerns. The configuration allows us to do exactly this, while making sure that there are no integrity violations along the way.

3.3 Using CIF in a team-based software development environment

The case study also enables us to illustrate how CIF can be applied in team-based software development environments. For example, consider a development environment that employs three major roles: i) a system architect, ii) a developer, and iii) a quality assurance officer. In such an environment, the *system architect role* would define the fundamental architecture of the application, plus CIF abstractions to constrain the subsequent definition and composition of the aspects envisaged in the design. In our particular case, this means defining the

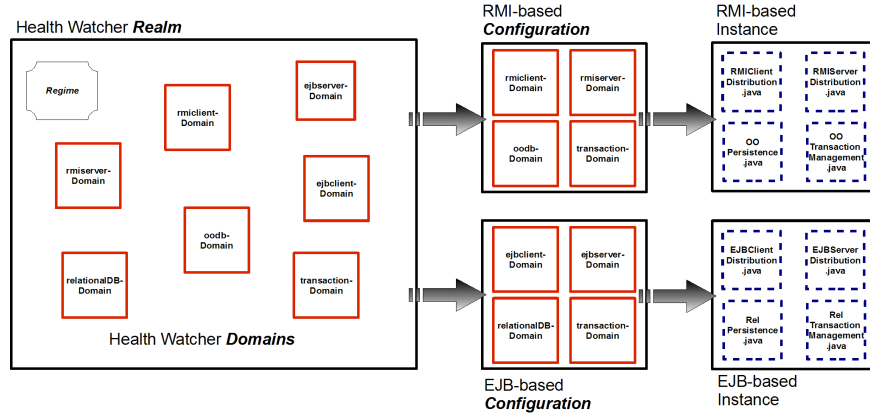


Fig. 10. Health Watcher with CIF

application’s basic client-server architecture, and its associated domains (e.g. `rmiclient-Domain` and `rmiserver-Domain`, plus domains for the persistence and transaction aspects as shown in Figure 10). The system architect would also constrain the way that these domains inter-relate by defining realms and associated regimes (e.g. in our case the system architect specifies that RMI and EJB communications are mutually exclusive and that transactional functionality is dependent on the presence of persistence functionality). The *developer* role would then implement the base modules and aspects identified by the system architect. When complete, the various modules and aspects would all be delivered to the *quality assurance officer* who would test the delivered aspects against their associated domains (using the CIF verifier; see Section 4) and define prototype configurations (such as our RMI- and EJB-based configurations as shown in Figure 10) that exercise the system in various application-specific ways. Finally, the quality assurance role would feed back the results of testing these configurations (such as integrity violations) for refinement to the systems architects and developers as appropriate.

4 Implementation

We have built a prototype implementation of CIF for the AspectJ language using AspectBench [36].

The implementation is logically structured in two parts: a static part and a dynamic part. The static part comprises a verifier that ensures that aspects conform to domains, and that realm and configuration specifications are self-consistent e.g. ensuring that any dependencies and mutual exclusion relations (specified using the *dependent()* and *exclusive()* integrity functions) are honoured. Our current implementation supports only “*closure(0)*” semantics – i.e. only top-level code is checked for conformance with regions (regions themselves are computed with assistance from the AspectBench weaver). Other

application-specific integrity functions can also be added. For example, in a team-based software development scenario where multiple architects collaborate to define domains, an *ignore(Region r)* function could be defined to demarcate an application-level region that should never participate in the definition of any domain.

The second part of the implementation is a dynamic verifier integrated with the AspectBench weaver, and uses callbacks to realise extensible weave-time checking behavior. Integrity functions that are integrated with this part are allowed to have side effects. For example, a weaver callback associated with the *precedence()* integrity function directs weaving to occur according to precedences specified by this function.

Figure 11 gives a layered view of our implementation. The *CORE* module implements the CIF’s central abstractions (domains, realms, and configurations) and performs the above-mentioned static consistency checks. Above this, the *CIFParser* module reads in and parses CIF specifications. The *CORE* and *CIF-Parser* modules are generic and entirely independent of any particular target AOP environment, the latter being represented primarily by the *Weaver* module. The generic parts are sheltered from the Weaver by semi-generic modules, namely the *AspectLang*, *LangParser*, and the *DynamicVerifier* modules: *AspectLang* provides an AOP-environment-specific adaptation layer between the *CORE* and the target AOP environment; *LangParser* takes care of parsing the programming language specific aspects that will participate in the composition; and *DynamicVerifier* provides the above-mentioned callback mechanism for integration with the weaver.

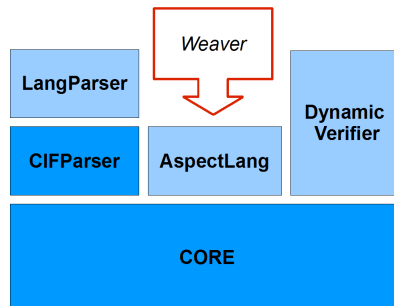


Fig. 11. CIF Implementation Architecture

We handle dynamic pointcuts (e.g. AspectJ’s *cflow*) in the following manner. Such pointcuts are commonly realised by inserting runtime checks (during compilation) at points (both for entry and exit control flows) where composition might occur [19, 6]. In that light, our strategy⁵ is simply to ensure that these checks are constrained to occur only within appropriately defined shadow

⁵ A possible alternative strategy could be to constrain only where composition should occur i.e. control flow exit points.

regions. This gives us an ‘upper limit’ for constraining dynamic composition without the need to be too restrictive. In future implementation work we plan to extend our exploration of runtime integrity support in dynamic environments. One key strategy here is to explore integrity functions that persist at runtime. For example, we could conceive of an integrity function that ensures that ‘around’ advice in a specified domain always calls `proceed()` [9, 29].

We also plan to provide an automated tool that would inspect all the aspects and pointcuts in an existing system and come up with corresponding domain specifications. This would relieve us of the burden of having to specify domains from scratch; it would provide an initial definition of regions for each aspect, which can be viewed and refined (at least by specifying integrity functions) as required. Complementary to this automated tool, we also plan to provide a GUI-based workbench to better evaluate the use of CIF in realistic software development environments.

5 Related Work

Although we argue that the literature has not yet comprehensively addressed the issue of integrity in AOP systems, we nevertheless acknowledge that a considerable body of work addresses a range of concerns within this general space.

In terms of the *where* dimension, a number of approaches attempt to restrict the shadows specified by pointcut expressions. For example, [8, 10, 11, 12] all offer ways to abstract the set of join points accessible to pointcut expressions so as to raise the semantic level and thereby increase the precision of the quantification. XPI [13] attempts to achieve similar ends through the use of an information hiding interface called ‘cross-cutting interfaces’. A complementary approach is adopted by [14] which offers a ‘negative’ pointcutting mechanism that allows sets of join points to be explicitly excluded from quantifications. In general, although this work enables greater control over the ‘where’ dimension, it does not at all address the ‘how’ or ‘what’ dimensions. Furthermore, all the proposed approaches lack an equivalent concept for domains and therefore restrict quantification in a coarse-grained manner: i.e. they do not allow *per-aspect* control over the quantification shadow. This is an important omission because shadow management is naturally a per-concern issue.

In terms of the *how* dimension, most work to date has focused on how multiple aspects woven at a common join point should relate to each other. For example, [17] and [18] provide a comprehensive catalogue of aspect interactions and suggest how unwanted interactions can be detected; [20] provides a systematic means of composing multiple aspects, and detecting and resolving potential conflicts; and [9] provides a domain-specific language to define execution constraints for aspect interactions. This work is largely complementary to ours in offering greater semantic depth in specific focus areas. However, it does not address the comprehensive range of ‘how’ issues (e.g. accommodating ordering issues, aspect instantiation semantics, or execution options) that we cover (see

Section 2). Our approach can also be viewed as an integrating framework for more focused work such as the above.

In terms of the *what* dimension, [21] discusses the notion of a permission system for aspects – i.e. to control which parts of a system a given aspect can access; however, no concrete mechanisms are proposed. Other work (e.g. [22, 23, 24, 25, 26]) attempts to classify and categorise advice (e.g. in terms of whether it is read-only or, if not, the degree of invasiveness permitted). Our approach, on the other hand (see Section 2), focuses on a finer-grained restriction of the behaviour that is applicable for a given advice type (i.e. in terms of the modules or libraries that can be used by advice). Furthermore, although the classifications provided by [22, 23] seem very general and widely applicable, they do not deal, as we do, with concern interactions (see Section 2). Also, although [24, 26] provide an intuitive classification, the reports generated by these tools require a lot of manual analysis, and each change requires the generation (and subsequent analysis) of a new report. Moreover, contrasting with [25], our approach does not require annotation of the underlying system. Finally, some research attempts to verify the behaviour of aspects in terms of static code analysis [27]. However this work (so far) only supports a limited source language, making it unsuitable for most real-world scenarios.

“Design by contract” approaches can also be viewed as relating to the ‘what’ dimension. For example, the approach proposed by [28] divides aspects into ‘observers’ and ‘assistants’ to facilitate modular reasoning; and [15] provides a similar classification of aspects. Also, [16] describes the notion of ‘aspect integration contracts’ which attempts to preserve integrity by equipping aspects with ‘required’ and ‘provided’ denotations. Although these approaches directly target the integrity issue (albeit only in the ‘what’ dimension) their downside is that they tend to be intrusive, verbose and difficult to manage. In contrast, CIF avoids cluttering source code by factoring integrity specification out of both base modules and aspects.

Finally, MAO [29] is an interesting system that addresses both ‘what’ and ‘where’ issues, but using an approach quite different from ours (see Section 2). In brief, they adopt a language-based annotation approach that enables programmers to explicitly characterise the effects of aspects. In terms of ‘what’, the programmer can, for example, write annotations that ensure that `proceed()` is called, that return values are returned unchanged, or that an advice may not change arguments. In terms of ‘where’, the programmer can constrain the quantification shadow of a given aspect. However, although MAO enables wide-ranging control over the integrity of an AOP system, it has significant disadvantages that mainly accrue from its language-based approach (again, compare our ‘factored out’ approach). In particular, MAO annotations are potentially complex and may need to be applied to base code as well as aspect code; moreover it is always difficult to find general acceptance of new language-level constructs.

6 Conclusion

As AOP gains more ground, it becomes increasingly important to provide a means to maintain a tight grip on the compositional flexibility it provides. We feel that the CIF approach offers a very promising way to achieve this, in a very practical manner.

We are encouraged with the results we have achieved so far, but we also realize that there is more work to be done. Perhaps the most significant limitation of our design that we have discovered to date is the predicate-based region definition approach. Experience has shown that this can be tricky to get right (especially using Java-inspired package predication) and may suffer from similar weaknesses as pointcuts – i.e. over strong or over weak predication. Currently, we are experimenting with alternative predication approaches and also with the integrity function facility to help minimise this weakness: for example, a ‘monitor’ function can keep track of complete enumeration (automatically, without any manual intervention) of programmatic elements picked out by specific regions and warn us when there are changes in this set.

More generally, we believe that our approach to integrity management is unique in not restricting or even changing the aspect language or the base application: CIF simply sits *alongside* these two components, to provide the necessarily integrity-preserving machinery. It thus cleanly separates out the integrity concern (or ‘meta-concern’). Furthermore, we believe that our proposal is complementary to several of the more partial approaches discussed in Section 5, and that these could in future work be integrated into our approach. For example, XPIs [13] can be applied alongside CIF at design time to clarify and facilitate the selection of appropriate regions and compositional styles. Similarly, it seems feasible to accommodate the execution constraints defined in [9] by encapsulating them in integrity functions. We can also include advice classifications similar to augmentation and replacement [24, 25, 26] in *advice-related-style*, thus extending them with compositional styles. Finally, it seems conceivable to relate contracts, as described by [16], to *domains* and to therefore provide stronger guarantees that aspects comply with concern-specific semantics.

References

1. Kiczales G. et al. Aspect-Oriented Programming. In *Proc. of ECOOP*, volume 1241 of LNCS, pages 220-242. Springer 1997
2. R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proc of Workshop on Advanced Separation of Concerns, OOP-SLA 2000*
3. David D. Clark and David R. Wilson. A Comparison of Commercial and Military Computer Security Policies. SP, p. 184, 1987 *IEEE Symposium on Security and Privacy*, 1987
4. Saltzer, J.H., Schroeder, M.D. The protection of information in computer systems. In *Proc of IEEE*, pages 1278-1308, 1975

5. Hilsdale, E., Hugunin, J. Advice weaving in AspectJ. *In Proc. of AOSD*, pages 26-35. ACM 2004.
6. Mezini, M. et al. Efficient control flow quantification. *In Proc of OOPSLA*, pages 125-138. ACM 2006
7. Stoerzer, M., Graf, J. Using pointcut delta analysis to support evolution of aspect-oriented software. *In Proc. of ICSM*, pages 653-656. IEEE 2005.
8. Kiczales, G., Mezini, Mira. Aspect-oriented programming and modular reasoning. *In Proc. of ICSE*, pages 49-58. ACM 2005.
9. Pawlak, R et al. CompAR: Ensuring Safe Around Advice Composition. *In Proc of FMOODS*, volume 3535 of LNCS, pages 163-178. Springer 2005.
10. J. Aldrich. Open Modules: Modular Reasoning about Advice. *In Proc. of European Conference on Object-Oriented Programming*, volume 3586 of LNCS, pages 144-168. Springer 2005
11. Gudmundson, S and Kiczales, G. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. *In Proc. of ECOOP*, 2001
12. Lieberherr, K et al. Aspectual Collaborations: Combining Modules and Aspects. *In Proc. of The Computer Journal*, volume 46 pages 542-565, 2003
13. Sullivan, K et al. Information hiding interfaces for Aspect-Oriented Design. *In Proc of ESEC/FSE-13*, pages 166-175. ACM 2005
14. David Larochelle et al. Join Point Encapsulation. *In Proc of SPLAT in conjunction with AOSD*, 2003
15. Lorenz, D.H., Skotiniotis, T. Extending Design by Contract for Aspect-Oriented Programming. *CoRR*, 2005
16. Lagaisse, B; Joosen, W, De Win, B. Managing semantic interference with aspect integration contracts. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect*, March 2004.
17. Sanen F. et al. Study on interaction issues. *Deliverable D44 of AOSD-Europe*, 2006
18. Katz E. et al. Detecting Interference Among Aspects. *Deliverable D116 of AOSD-Europe*, 2006
19. Dinkelaker T. et al. Inventory of Aspect-Oriented Execution Models. *Deliverable D40 of AOSD-Europe*, 2006
20. R. Douence, P. Fradet, M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. *In Proc of AOSD*, pages 141-150. ACM 2004
21. Bart De Win and Frank Piessens and Wouter Joosen. How secure is AOP and what can we do about it? *In Proc of SESS*, pages 27-34. ACM 2006
22. M. Sihman and S. Katz. Superimposition and aspect-oriented programming, *The Computer Journal*, 46 (5), 2003, pp. 529-541
23. Dantas, D., Walker, D. Harmless Advice. *In Proc. Of POPL*, pages 383-396. ACM 2006
24. Rinard, M. et al. A classification system and analysis for Aspect-Oriented Programs. *In Proc of SIGSOFT*, pages 147-158. ACM 2004
25. Munoz, F., Baundry, B., Barais, O. Improving Maintenance in AOP Through an Interaction Specification Framework. *In Proc of ICSM*, pages 77-86. 2008
26. Zhang, D., Hendren, L., Static Aspect Impact Analysis. abc Technical Report, 2007
27. Krishnamurthi, S et al. Verifying aspect advice modularly. *In Proc of SIGSOFT*, pages 137-146. ACM 2004.
28. Clifton, C, Leavens, G. Observers and assistants: A proposal for modular aspect-oriented reasoning. *In Proc of FOAL Workshop*, 2002

29. Clifton, C et al. MAO: Ownership and Effects for More Effective Reasoning About Aspects. *In Proc of ECOOP*, pages 451-475. Springer 2007.
30. Alexander, R.T., McEachen, N; Distributing classes with woven concerns: an exploration of potential fault scenarios. *In Proc of AOSD*, pages 192-200. ACM 2005
31. S. Soares, P. Borba, E. Laureano. Distribution and Persistence as Aspects. *Software: Practice and Experience*, 36(6), 2006.
32. Greenwood, P. et al. On the Contributions of an End-to-End AOSD Testbed. *In Proc of Early Aspects at ICSE*, IEEE 2004.
33. The AspectJ Project, <http://www.eclipse.org/aspectj>
34. JBoss AOP, <http://www.jboss.org/jbossaop>
35. H. Rajan, K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. *In Proc of ICSE*, pages 59-68. ACM 2005.
36. The AspectBench Compiler for AspectJ, <http://abc.comlab.ox.ac.uk>