Chainmail: Defensively Consistent Modular Specifications for an Open World

Sophia Drossopoulou[1], James Noble[2], Toby Murray[3], Mark S. Miller[4]

[1]Imperial College London, [2]Victoria University Wellington, [3]DATA61/UNSW, [4]Google Inc.

Abstract

Most specification languages labour under a monolithic, closed-world assumption: that all the modules comprising a program, and all their interactions, can be known ahead-of-time. Specifications are tied tightly to the code of the modules that implement them. Many contemporary systems, however, must live in an open world: where there is no central trusted authority that can validate components. Open world systems comprise modules developed by many different parties, linked together dynamically in unforeseen constellations, and have to function correctly under attacks external modules which may be malicious.

In this paper we propose *Chainmail*, a modular, defensive, specification language for the open world. *Chainmail* specifications are modular, as separate concerns in a system can be captured as individual specifications or policy definitions which can cross-cut multiple modules. *Chainmail* policies define not only method pre- and post- conditions, but also give invariants that must be maintained *irrespective* of any other code in the system, even when the actual composition of the system is unknown. Taken together, this means that *Chainmail* can specify modules that must be *defensively consistent*, guaranteeing their integrity in an arbitrary open world. We present four small case studies of *Chainmail* specifications, and give a formal definition of *Chainmail*'s semantics.

Categories and Subject Descriptors D.2.1 [*Requirements / Specifications*]: Languages; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.4.6 [*Security and Protection*]: Verification

1. Introduction

Contemporary open systems are made up of modules developed by many different parties, linked together dynamically in unforeseen constellations. Modules have to function correctly, even when they collaborate with external modules of unknown provenance. In practice this means modules must work defensively to protect their own modules' integrity when they interact with potentially malicious external code.

Not only is the development of open code error prone, but it is also difficult to specify correctly [10, 11]; verifying that a particular module maintains the resulting specifications in an open environment is even more difficult, because the specifications need to describe the behaviour of the code both for those few secure systems when all objects are trustworthy, but for the many insecure systems when one or more unknown external objects may be malicious. To misparaphrase Tolstoy, secure systems are all alike; every insecure system is insecure in its own way [53]. Attackers *"only have to be lucky once"* while secure systems *"have to be lucky always"* [2].

The traditional (philosophically, the "modern" approach) to construct secure systems is to build up trusted components as an hierarchy on top of a known and trusted secured computing base, underpinned by verified and verifying compilers [21]. This approach relies on a closed world assumption: there is an explicitly demarcated border between the inside and the outside of the system, and the whole system can be trusted because (and as long as) each component inside the border can be trusted. Similarly, each module is an independent demesnes [54], jealously protecting its own invariants behind its own borders, giving rise to a series of layers of trust, layered virtual machines [8] or protection rings [48].

Open systems, on the other hand, must live under an open world assumption: they are composed of a wide range of component objects with different levels of mutual trust (or distrust). Rather than all components within an explicit boundary being trusted equally, an open system is a post-modern heterarchy, a rhizome, a dynamic arrangement of objects each of which may trust different combinations of objects in different ways [42]. Rather than strict boundaries between the outside of the system and the various layers inside, an open system is demarcated implicitly, as objects dynamically come into relationships with other objects, as those objects mutually interact, and as those relationships eventually dissolve. Trust relationships may be established gradually, as dynamic interactions between components let individual components determine which other components they are willing to trust (and which they are not). In an open world, it is crucial that code fails safely: each module must take care that it can protect its own invariants, especially when a interacting with untrusted modules.

The critical problem with building systems for an open world is that the actual invariants that must be maintained by a program will be *implicit*, scattered throughout the program's modules. Any part of a program that uses an object from a trusted module may (by oversight, error, or fraud) hand that object to an untrustworthy module, giving potentially malicious code access to all the services provided by the trustworthy module [4, 22]. This makes it hard to determine what invariants are actually maintained by a given module, and thus verify the integrity of the overall system. Miller [35, 36] defines the necessary approach as **defensive consistency**: *"An object is defensively consistent when it can defend its own invariants and provide correct service to its well behaved clients, despite arbitrary or malicious misbehaviour by its other clients."* Defensively consistent modules are particularly hard to design, to write, to understand, and to verify: but they have the great advantage that they make it much easier to make guarantees about systems composed of multiple components [41].

In this paper, we present a modular specification language, *Chainmail*, that is designed to support defensively consistent specifications of these kinds of open systems. Building on the object-capability model [37], *Chainmail* specifications are modular, as separate concerns in a system can be captured in as individual specification or policy definitions. *Chainmail* specifications not tied to any particular module in a system but can define the behaviour of any complying module, and can cross-cut multiple modules. *Chainmail* policies define not only method pre- and postconditions, but also give invariants that must be maintained *irrespective* of any other code in the system, even when the actual composition of the system is unknown. Taken together, this means that *Chainmail* can specify modules that must be *defensively consistent*, guaranteeing their integrity in an arbitrary open world.

Contribution This paper extends earlier informal work mostly presented at workshops [10, 11, 13, 43]. Here we present the full design of *Chainmail* for the first time, and show how *Chainmail* allows us to write modular, multidimensional specifications for the open world. As is traditional, some details are relegated to a technical report [14].

To address the open nature of the systems, we make the meaning of policies parametric with any code that may be linked to the current system. We give examples to show how *Chainmail* modules can be written in a very robust, defensively consistent manner, so that no further, malicious code can steal their secrets or break their integrity.

Our design of *Chainmail* is part of a larger research project about specifying and verifying systems in an open world. Security is of particular concern in open systems, and *Chainmail* includes assertions such as (**obeys** , $\mathcal{M}ay\mathcal{A}ffect$, and $\mathcal{M}ay\mathcal{A}ccess$) to allow us to reason explicitly about module's security properties and guarantees. Elsewhere we demonstrate how these constructs let us describe how components can coöperate to establish trust gradually, and to delineate the risks involved in that coöperation [15]. Security, trust, and risk are not the focus of this paper, rather here we concentrate on the core features of *Chainmail*, showing how *Chainmail* policies permit modular, defensively consistent specifications.

To give a semantics to *Chainmail* policies we have also defined a core object-oriented programming language, $\mathcal{F}ocal$, (the Featherweight Object Capability Language, not to be confused with FOCAL [33]), described in more detail elsewhere [14]. $\mathcal{F}ocal$ is a simple, dynamically typed language with traditional classes as its modularly mechanism. That is, *Chainmail* provides separation of concerns for *specifications* of object-oriented *implementations*. We made this design choice for several reasons, primarily because we are interested in specifications, rather than implementations. We are primarily interested in specifying "real world" programs, which tend to be written in OO languages like JavaScript, Java or E, rather than e.g. aspect- or subject- oriented languages; and that we wanted to keep $\mathcal{F}ocal$ as simple as possible. We hope to extend our specification techniques to more modular languages in future work.

To prove a program's adherence to *Chainmail* specifications, we have developed a Hoare logic and associated inference rules. We do not discuss the logic or these rules in this paper, although they are present in the technical report [14].

Disclaimers Throughout this paper, we make the simplifying assumptions that no two different arguments to methods are aliases, that the program is executed sequentially, that we can quantify over the entire heap, that objects do not breach their own encapsulation or throw exceptions, that machines on open networks are not mutually suspicious, and that any underlying network is error-free. This allows us to keep the specifications short, and to concentrate on the questions of risk and trust. Aliasing, concurrency, quantification, confinement, network errors, and exceptions can be dealt with using known techniques, but doing so would not shed any further light on the questions addressed here.

Paper Organization The next section informally introduces the *Chainmail* specification language, and then section 3 presents four small case studies of *Chainmail* specifications showing how they support defensively consistency. Section 4 then presents a formal definition of the core of

Chainmail, section 5 discusses related work, and section 6 concludes.

2. *Chainmail*

In this section we introduce our specification language *Chainmail*, and describe how it can be used to write specifications for defensively consistent modules. We define *Chainmail* formally below in section 4.

Chainmail specifications are modular because we specify a system as a series of *Chainmail* `specification` modules, rather than using a single monolithic specification for an entire system. Each *Chainmail* specification module consists of number of named `policy` clauses. By convention, *Chainmail* specifications are named in `CamelCase` while policies are named beginning with `Pol` have words separated by `Pascal_Style_Underscores`.

Chainmail specifications and policies overlap and are interlinked to provide strong protection against attacks — just like the links in physical chainmail. Each policy should aim to capture one very specific concern in the design of a system: policies may be pre- and post- conditions on method calls; or one-state or two-state invariants that a module that conforms to the specification must maintain even though any other code may be executed, including code that may use the module being specified.

Specifications can write `o` **obeys** `Spec` as an assertion that some object `o` conforms to the specification `Spec`. We have introduced **obeys** elsewhere to deal with trust; here we will show how **obeys** can also support modularity by tying specifications together as necessary. We also use **obeys** to decouple specifications from any particular implementation classes. This is important in an open world, where we cannot constrain the provenance of implementation of objects if we wish to maintain openness. The **obeys** assertion constrains or qualifies only over the actual behaviour of an object, not the class to which it belongs, or any interfaces or traits it is declared to implement.

Chainmail specifications may use `this` to refer to the object begin specified. Using `this` in a specification `S` implicitly requires that `this` **obeys** `S`.

Chainmail also offers assertions about the shape of the heap and of potential effects caused by objects. The assertion $\mathcal{M}ay\mathcal{A}ffect$`(o,p)` means that it is possible that some method invocation on `o` would affect the object or property `p`, while $\mathcal{M}ay\mathcal{A}ccess$`(o,p)` means that it is possible that the code in object `o` could potentially gain a reference to `p`. In practice, $\mathcal{M}ay\mathcal{A}ccess$`(o,p)` means that `p` is in the transitive closure of the points-to relation on the heap starting from `o` including both public and private references.

The **obeys**, $\mathcal{M}ay\mathcal{A}ffect$, and $\mathcal{M}ay\mathcal{A}ccess$ predicates were introduced in our work on Risk and Trust in object-capability programs [11, 13, 15]. The contribution of this paper is the definition of *Chainmail*, and its general use in specifying defensively consistent modules. For completeness, we discuss trust and risk by revisiting the Purse case study in section 3.4.

Chainmail policies (and specifications) can cross cut both each other and the various modules and objects in the system being specified. The validity of a specification is the conjunction of its policies; a module or an object must satisfy all the specification's policies for us to consider that the object meets the specification. Policies and specifications are not tied to any specific module or class: rather, any implementing module that satisfies the specification's policies obeys the specification. Chainmail has a many-to-many relationship between policies and implementations, explicit embodied in the **obeys** assertion.

Considering the separation of concerns between specifications and code, *Chainmail* supports both obliviousness and quantification. *Chainmail* specifications require no changes or annotations to code being specified (which is thus oblivious to the specifications) and by applying to different implementations of methods or classes, *Chainmail* can quantify over the base program and its execution. *Chainmail*'s quantification and join point model is quite simple, however: one-state invariant policies quantify over each visible state of the program; two-state invariants (`any_code` policies) quantify over pairs of successive visible states, while policies giving invocation pre- and post-conditions quantify over the pre- and post- states of any matching method invocation in any class or module.

2.1 Specifications and Implementations

Figure 1 shows a simple implementation of a `counter` class in a simple untyped object-oriented language. The `tick` method increments the counter, and the `count` method retrieves its value.

```
class counter {
  var value := 0
  method count {value}
  method tick {value := value + 1}
}
```

Figure 1. Simple Counter Example

Figure 2 shows a *Chainmail* `ValidCounter` specification that could apply to the `counter` class. This specification consists of one ghost field current (a ghost field may appear only in specifications) and four policies.

The first policy, `Pol_new_ValidCounter` specifies the `new counter` expression that creates the new counter instance. First, the result `res` must itself conform to the `ValidCounter` specification (`res` **obeys** `ValidCounter`). Second, we assert that ghost `current` field for the new instance must have the value zero. Finally, we require that no pre-existing object may have a reference to the newly-created counter.

The second and third policies are straightforward. `Pol_count` ensures that the `count` method returns the current value of the counter, and `Pol_tick` ensures that the `tick` method increments the count.

```
specification ValidCounter {
  field current // Number

  policy Pol_new_ValidCounter
      true
         { res = new counter }
      (res obeys ValidCounter) ∧
      (res.current == 0) ∧
      ¬∃ o : MayAccess_pre(o,res)

  policy Pol_count
      true
         { res = this.count }
      res == this.current

  policy Pol_tick
      true
         { this.count }
      this.current == this.current_pre + 1

  policy Pol_protect_count
      ∀ o,c:Object. c obeys ValidCounter ∧
          MayAffect(o,c.count) → MayAccess(o,c)

  policy Pol_count_increases
      true
         { any_code }
      this.current >= this.current_pre
}
```

Figure 2. ValidCounter specification

The fourth policy is a one-state invariant. Pol_protect_count guarantees that a valid counter c's current count can only be changed — *MayAffect*(o,c.balance) — by an object o that may access that counter: *MayAccess*(o,c).

The final policy, Pol_count_increases, requires that the counter may only increase. This is an any_code policy, that is, a two-state assertion which must hold between any two two states in a program's execution, irrespective of the module to which that code belongs.

We argue the counter class in Figure 1 satisfies the ValidCounter specification in Figure 2 as follows. The semantics of the underlying language (variable initialisation, classes creating new unique objects) ensure the Pol_new_ValidCounter policy is satisfied; and the straightforward method bodies satisfy Pol_count, Pol_tick, and also Pol_count_increases. Finally we can argue that Pol_protect_count is satisfied because there is no exposure of this in the body of counter.

This argument relies on Meyer's open-closed principle [34]. Even in open systems, where modules may be added in to a system at any time, and where the interpretation of specifications are necessarily in an open world, we assume that individual implementation modules — here, classes — are closed: they have a fixed, stable, definition, and can potentially be validated against one or more specifications.

2.2 Defensive Consistency

Pol_protect_count, in particular, will require defensively consistent programming in any implementing module to ensure that it is satisfied. Pol_protect_count cannot be expressed as a simple invariant because it is about a relationship between two states, rather than the admissibility of a single state. Nor can Pol_protect_count be expressed as traditional method pre- and post-conditions, because it must apply to all modules that implement the specification, and pre- and post-conditions can only constrain the behaviour of methods actually described in the specification. An implementing module could have other methods that are not mentioned in that specification.

Because *Chainmail* assumes an open environment, we assume any other code may be linked together with the module implementing the ValidCounter specification to make the final system: and indeed, any module that implements ValidCounter may also implement any number of other specifications in addition to ValidCounter. The interpretation of *Chainmail* specifications is not that the resulting system *does not* breach the policies, but rather, that the resulting system *cannot* breach the policies.

This difference reflects critically different underlying assumptions between closed and open systems. Closed systems can depend on coöperation between modules, where all the modules in a system work together to maintain all of their invariants, so ensuring none are breached. This is possible only when all the modules in the system are known in advance, and known to be trustworthy in advance, that is, in a closed system. In contrast, in an open system, it is impossible to know all of the modules that may be linked together to make any given system configuration, and impossible to rely on the goodwill of the other modules in the system to coöperatively maintain invariants. Rather, modules in a open system must look out for themselves: other modules may actively try to break their invariants, steal their data, subvert their invariants: in short, to ensure that whatever the rest of the system tries to do, its own policies will be followed.

This is the core of defensive consistency: no matter what the rest of the system may or may not do, a module that implements this specification must ensure that the system as a whole will satisfy all its policies — including any_code invariants — even in the face of any other code executed anywhere else in the system.

Miller's original definition of defensive consistency (*"a defensively consistent object will never give incorrect service to well-behaved clients"* [35]), and Murray's subsequent formalisation in the logic of causation [41] are both intensional: specifying the essential properties of a defensively consistent object, but do not describe how objects should be specified or designed in order to meet that criterion. When writing in *Chainmail*, we will use following practical guidelines to check for defensive consistency:

($\mathcal{A}$) **Explicit Trust:** all objects used by a specification must be defensively consistent.
($\mathcal{B}$) **Fail safely:** treat success and failure explicitly.
($\mathcal{C}$) **Complete specifications:** cover all cases:

($\mathcal{C}$.1) — method pre- and post-conditions.
($\mathcal{C}$.2) — invariants between method calls.
($\mathcal{C}$.3) — invariants across other objects.
We use these guidelines to analyse the case studies we present in this paper.

3. Case studies

3.1 Modular Implementations

Figure 3 shows another implementation of the `ValidCounter` from Figure 2 — demonstrating that specifications are not tied to particular implementations

```
class loggingCounter {
  def log = col.list
  method tick { log.push(sys.currentTime) }
  method count { log.size }
}
```

Figure 3. Logging Counter Example

Once again it is relatively straightforward to demonstrate that this class meets the specification. Again, the semantics of classes and the behaviour of the `col.list` list collection class ensure the `Pol_new_ValidCounter` policy is satisfied. The `count` and particularly the `tick` methods (which adds the time it is called to a log) are more complex than the `counter` versions but validation will still be relatively straightforward. Because nothing is removed from the `log`, the size of the log cannot decrease, and this plus the definition of the `count` method ensures `Pol_count_increases`. We can validate `Pol_protect_count` similarly ($\mathcal{C}$).

Reading the Log

The `loggingCounter` in Figure 1 unfortunately doesn't allow any way to access the times stored in the log. To solve this problem, we can add another method to get that log.

```
  method getLog {log}
```

Unfortunately this implementation breaks the `Pol_protect_count` policy of the `ValidCounter` specification. To see why, consider the following code:

```
  def c = counter
  def l = counter.log
  l.removeAll
```

This code will reset `c.count` to zero — and does so via an object, `l` that has no access to `c`, thus breaching `Pol_protect_count` which says that access to `c` is required to change it. The problem here is *representation exposure* [7]: the problem occurs precisely because the counter's representation can be accessed independently of the larger `loggedCounter` implementation of which it is a part.

Defensive Consistency

Note that traditional access protection, e.g. ensuring that the `log` field in the `loggedCounter` class is private and so cannot be accessed that classes is not enough to avoid this problem, because such protection does not prevent a public method, like the `getLog` accessor above, from reading the field and handing the object contained in that field to any other object that asks ($\mathcal{C}$.3). More advanced language protection mechanisms such as ownership types [5] can prevent this particular problem.

This is where defensive consistency is crucial: to meet the specification programmers must ensure the integrity of their objects cannot be undermined, no matter what external objects that somehow come into contact with them may do. In this case, there is a simple fix — copying the log before it is returned:

```
  method getLog {log.copy}
```

Returning a copy of the log prevents the result of `getLog` begin used to affect the value of the `count`, and so a `loggingCounter` extended with this version would once again meet the `ValidCounter` specification.

3.2 Subject-Observer

The Observer pattern, also known as Subject-Observer, is one of the most well-known of Gamma et al.'s *Design Patterns*. As with many design patterns, Observer is a cross-cutting relationship between several classes in an object-oriented design. For these reasons, aspect-oriented or other multidimensional modelling techniques have proved effecting for modelling design patterns [17].

Figure 4 shows a version of the Observer pattern. Each role in the pattern — `subject` and `observer` — has their own class. As is common when modelling Observer, for space reasons we simply arrange to update the observer's `observation` variable whenever the subject's `data` changes.

```
class subject {
  def observers =  col.set
  var mydata := 0  // private by default
  method data {mydata}
  method change(newdata) {
    mydata := newdata
    for (observers) do { o -> o.notify }
  }
  method addObserver(o) {
    if (this == o.subject)
      then {observers.add(o)}
  }
}

class observer(mySubject) {
  method subject {mySubject}
  var observation := subject.data
  method notify { observation = subject.data }
}
```

Figure 4. Subject and Observer

Figure 5 shows the core of the specification for observer pattern. Here we have two specifications that work together: `ValidSubject` for the subject, and `ValidObserver` for

```
specification ValidSubject {
  policy Pol_subject_observer
    ∀ o ∈ this.observers.
          o obeys ValidObserver ∧
                o.subject == this
}
specification ValidObserver {
  policy Pol_observation
    this.observation == this.subject.data
}
```

Figure 5. Specification of `ValidSubject` and `ValidObserver`

the observer. The observer's policy specifies the key invariant: that an observer's observation must equal its subject's data ($\mathcal{C}$.3). The subject's policy establishes a configuration of the system where the observer's policy can make sense: all the objects contained a `ValidSubject`'s list of observers must obey the `ValidObserver` specification ($\mathcal{A}$), and they must must be observing that particular subject. Note that while the `Pol_subject_observer` policy explicitly requires requires `o` **obeys** `ValidObserver` there is no explicit requirement that subjects are valid. Rather, that requirement is implicit in the specification, because `this` in a policy implicitly **obeys** the specification containing that policy.

Defensive Consistency

Using the subject and observer is simple: we create an instance of the subject, modelling a weather station, say; an observer on the weather station; and then add the observer to the subject:

```
def weatherStation := new subject
var weatherWebPage := new observer(
    weatherStation)
weatherStation.addObserver(myWebPage)
```

At this point, whenever the weather station gets a change in the weather:

```
weatherStation.change("Otherwise Fine")
```

its observers will be notified, maintaining `Pol_observation`. The implementation in figure 4 is defensively consistent, because this relationship will be maintained no matter what code in other modules may try to do to the subject or observer ($\mathcal{C}$.3). In particular, the `observer` class is carefully designed so that its `observation` field cannot be modified from outside — code attempting to change an observation directly, e.g.:

```
weatherWebPage.observation := "Rain"
```

must be prevented by programming language mechanisms, typically dynamically (e.g. in JavaScript or E) but potentially statically (Java's class verifier). Similarly this design depends on the subject's `data` field only be updated by calling the `change` method: we assume all fields are only accessible via "`this`" unless explicitly annotated `public` ($\mathcal{C}$.1).

```
specification FileSystem {
  policy Pol_getFile
    name ∈ String
          { res = this.getFile(name) }
    (res == nil) ∨
    ((res obeys File) && res.isOpen == false)
}

specification File {
  field isOpen // Boolean

  policy Pol_open
    true
        { this.open }
    this.isOpen == true

  policy Pol_read_1
    true
        { res = this.read }
      (res ∈ String)  ⟶  this_pre.isOpen

  policy Pol_read_2
    true
        { res = this.read }
     (res == nil)  ⟶  \neg this_pre.isOpen

  policy Pol_close
    true
        { this.close }
    this.isOpen == false
}
```

Figure 6. Specification of `FileSystem` and `File`

3.3 Files

Our third case study considers a simple file interface. Figure 6 shows two related specifications: `FileSystem` which supports navigation in a file system, and `File` that supports reading a file.

The `FileSystem` specification has only one policy `Pol_getFile` which defines the `getFile` method. This method accepts a string argument, and either returns `nil` (presumably if that string does not denote a file) or a closed `File` object.

The `File` specification in Figure 6 supports opening, reading, and closing the file. `Pol_open` and `Pol_close` open and close the file respectively. `Pol_read_1` specifies an attempt to read an opened file: the read method returns a `String` containing the file contents. `Pol_read_2` specifies the situation when an attempt is made to read a closed file: once again `nil` must be returned.

Defensive Consistency

The specifications in Figure 6 quite restrictive, especially as they specify explicitly the results that are returned when methods fail (as in `getFile` and `read`). `Pol_read_1` and `Pol_read_2` both specify the same method call (on

```
specification ValidPurse {
  field balance // Number

  policy Pol_deposit_1     //   success case:
      amt∈ ℕ
         { res = this.deposit(amt, src) }
      res → (
              src obeys_pre ValidPurse ∧ CanTrade(this,src)_pre
              ∧ 0≤amt≤src.balance_pre ∧
              this.balance=this.balance_pre+amt ∧ src.balance=src.balance_pre−amt  ∧
              ∀p.(p obeys_pre ValidPurse ∧ p∉ {this,src} → p.balance=p.balance_pre)  ∧
              ∀o:_pre Object. ∀ p obeys_pre ValidPurse.  MayAccess(o,p) → MayAccess_pre(o,p)  )

  policy Pol_deposit_2     //   failure case:
      amt∈ ℕ
         { res = this.deposit(amt, src) }
       ¬res → (
              ¬( src obeys_pre ValidPurse ∧ CanTrade(this,src)_pre ∧ 0≤amt≤src.balance_pre) ∧
              ∀p.(p obeys_pre ValidPurse→ p.balance=p.balance_pre)   ∧
              ∀o:_pre Object. ∀ p obeys_pre ValidPurse.  MayAccess(o,p) → MayAccess_pre(o,p)  )
```

Figure 7. Specification of `ValidPurse`

`read`): but `Pol_read_1` specifies the successful case, and `Pol_read_2` the failure case ($\mathcal{B}$).

These specifications again demonstrate the necessity of defensively consistent specifications and implementations for open systems — specifications and implementations that do not assume that clients will coöperate with the implementations to meet the specifications. We see this as conditions such as `this.isOpen` appear in policies' postconditions, rather than their preconditions ($\mathcal{C}$.1).

Consider the following Eiffel-style Design by Contract alternative policy for reading a file:

```
  policy Pol_read_Eiffel
    this.isOpen
       { res = this.read }
    res ∈ String
```

in `Pol_read_Eiffel`, the test that the file is open is handled by the precondition, and the result is guaranteed to be a string. This is fine in closed system with an underlying assumption that modules will all coöperate to e.g. ensure that the invariants of the `File` module are not subverted. The problem with this kind of specification in a open system is that it does not explain what should happen when the precondition is breached: the behaviour of a *closed* `File` object that receives the `read` method is (intentionally) unspecified. Paraphrasing Bertrand Meyer, closed systems can adhere to the principle that *"under no circumstances shall the body of a routine check for the precondition of the routine"*, and as a consequence *"if the class does not satisfy precondition, then the class is not bound by the post condition. . . . [it] can either throw an exception or return a wrong result."*. [34, 52].

Our specifications illustrate why defensive consistency is the antithesis of this principle: a defensively consistent specification (and hopefully a defensively consistent implementation of that specification) must be designed for an open system where such a coöperative assumption is untenable. Other modules may well attempt to e.g. call `getFile` on secure system files they should not be able to access, or call `read` on files that have already been closed; our specifications must give accurate guidance to implementations about how those situations must be handled, rather than "returning a wrong result" which could include a `read` method returning private information that was meant to be kept secure. Eiffel style specifications, designed for closed systems, explicitly permit these implementations which can be disastrous for open systems: the *Chainmail* style, in contrast, encourages full specifications of both correct and incorrect invocations.

3.4 Purses

As a final case study we revisit Miller's Purse example which we have described in more detail elsewhere [11, 13, 15].

Figure 7 shows two policies from the `ValidPurse` specification. A Purse is a model of a store of value, such as a bank account. Writing `dst.deposit(amt, src)` will either transfer `amt` from the `src` purse to the `dst` purse and return true, or do nothing and return false. Taken together, the `Pol_deposit_1` and `Pol_deposit_2` policies specify this behaviour. The postconditions of the policies do all the work. If the argument `src` is a valid purse ("'src **obeys** ValidPurse") that can trade with the receiver, and has sufficient balance, the transaction goes ahead and returns `true`, otherwise the transaction does not go ahead and returns `false`.

Defensive Consistency

This specification ensures defensive consistency in a number of ways. Most obviously, the `Pol_deposit_1` and `Pol_deposit_2` policies specify both success and failure cases in detail: the purse does not rely on its clients "playing nicely" ($\mathcal{B}$). For this reason, rather than encoding sufficient conditions in the precondition, we specify necessary conditions in the postcondition, and make them conditional on the return value.

Secondly, the specification does not relies on e.g. class membership to validate the `src` purse. Rather, we specify that the `src` purse **obeys** this specification, which makes no claim about the particular class of the `src` object ($\mathcal{A}$). We also use a local (abstract) predicate `CanTrade` between individual purses (e.g. if they represent accounts at the same bank). An open system could have many different "families" of purses, different implementations of this specification, and only purses from the same family would be able to trade with each other. The specification does not rely on a type system or some other central authority to know which purses are trustworthy and which are not, rather it uses **obeys** to make that relationship explicit.

Thirdly, we carefully frame the behaviour of other valid purses to avoid unexpected or unwanted effects ($\mathcal{C}.3$). For a purse to be valid, calling `deposit` must not somehow affect other purses elsewhere. If the transaction is unsuccessful, no purses may be affected; if successful, only `this` and `src` can change. Similarly, we require `deposit` cannot leak access to any purse: if after the method call, a pre-existing `o` has access to a `ValidPurse` object `p`, then `o` must already have had access to a `p` before the call.

4. Formal Model

In this section define the core of the *Chainmail* specification language. While we have discussed *Chainmail* elsewhere [13, 15] in this paper we provide a formal definition for the whole core of the specification language. Many details, including the featherweight programming language $\mathcal{F}ocal$ upon which it relies, and our Hoare Logic, are presented in our technical report [14]; to facilitate cross-references, we adopt its numbering for definitions.

Chainmail specifications are a conjunction of a set of named policies. *Chainmail* policies are based on one-state assertions (A) and two-state assertions (B). To express the state in which an expression is evaluated, we annotate it with a subscript. For example, $x > 1$ is a one-state, and $x_{pre} - x_{post} = 1$ is a two-state assertion. Validity of an assertion is defined in the usual manner, *e.g.* in a state σ with $\sigma(x) = 4$ we have $M, \sigma \models x > 1$. If we also have $\sigma'(x) = 3$, then we obtain $M, \sigma, \sigma' \models x_{pre} - x_{post} = 1$. *Chainmail* specifications may also express ghost information, which is not stored explicitly in the state σ but can be deduced from it — e.g. the length of a null-terminated string.

4.1 Expressions and Assertions

We first define expressions, *Expr*, and assertions A, which depend on *one state* only. We allow the use of mathematical operators, like $+$ and $-$, and we use the identifier *f* to indicate functions whose value depends on the state, e.g. the function `length` of a list. Functions model ghost fields. Arguments *Arg* are access paths within the underlying language [14]. The difference between expressions and arguments is that expressions may express ghost information, which is not stored explicitly in the state σ but can be deduced from it — e.g. the length of a list that is not stored with the list.

Definition 8 (Expressions)**.**

$$\begin{array}{lll} \textit{Expr} & ::= & \textit{Arg} \ \mid \ \textit{Val} \ \mid \ \textit{Expr} + \textit{Expr} \ \mid \ ... \\ & \mid & \textit{f(Expr}^*\textit{)} \\ & \mid & \textbf{if } \textit{Expr} \textbf{ then } \textit{Expr} \textbf{ else } \textit{Expr} \\ \textit{funDescr} & ::= & \textbf{function}\, \textit{f(ParId}^* \textit{)} \ \{ \ \ \textit{Expr} \ \ \} \end{array}$$

We now define the values of such expressions, and the validity of one-state assertions as follows:

Definition 9 (Interpretations)**.** *We define the interpretation of expressions,* $\lfloor \cdot \rfloor$: *Expr* $\times$ *Module* $\times$ *state* $\rightarrow$ *Value using the notation* $\lfloor \cdot \rfloor_{M,\sigma}$*:*

- $\lfloor val \rfloor_{M,\sigma}$ = val*, for all values* $val \in$ *Val.*
- $\lfloor a \rfloor_{M,\sigma}$ = $\lfloor a \rfloor_{\sigma}$*, for all arguments* $a \in$ *Arg.*
- $\lfloor e_1 + e_2 \rfloor_{M,\sigma}$ = $\lfloor e_1 \rfloor_{M,\sigma} + \lfloor e_2 \rfloor_{M,\sigma}$.
- $\lfloor f(e_1, ...e_n) \rfloor_{M,\sigma}$ = $\lfloor Expr[e_1/p_1, ...e_n/p_n] \rfloor_{M,\sigma}$ *where* $M(f) = \textbf{function}\, f\ (\ p_1 ... p_n)\ \{\ \ Expr\ \}$*,* *undefined, otherwise.*
- $\lfloor \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \rfloor_{M,\sigma}$ $= \lfloor e_1 \rfloor_{M,\sigma}$*, if* $\lfloor e_0 \rfloor_{M,\sigma}$=**true**, $= \lfloor e_2 \rfloor_{M,\sigma}$*, if* $\lfloor e_0 \rfloor_{M,\sigma}$=**false**. *and undefined, otherwise.*

One-state assertions We now define a language of assertions which depend on one state. Most of these are conventional: expressions and relations from the underlying base language; specification level binary predicates; existential and universal quantification, and the assertion $\textit{Expr}\colon\textit{ClassId}$ which expresses class membership.

We also introduce three specific assertions types to assist in reasoning about security: $\mathcal{M}ay\mathcal{A}ffect$ and $\mathcal{M}ay\mathcal{A}ccess$ model risk, while *Expr* **obeys** *SpecId* models trust. These assertions are *hypothetical*, in that they talk about the potential effect of execution of code, or of the existence of paths to connect two objects. The $\mathcal{M}ay\mathcal{A}ffect$ assertion ascertains whether its first parameter may execute code which affects the second one; the $\mathcal{M}ay\mathcal{A}ccess$ assertion ascertains whether its first parameter has *any* path to the second one; and **obeys** captures an assumption that an object conforms to a given specification. These three assertions are motivated and discussed in depth elsewhere [15]: we include their definitions here for completeness.

Definition 10 (One-state Assertions)**.**

$$
\begin{array}{lcl}
A & ::= & \mathit{Expr} \;\mid\; \mathit{R(Expr^*)} \\
 & \mid & \mathit{Expr} \geq \mathit{Expr} \;\mid\; A \wedge A \;\mid\; \ldots \\
 & \mid & \exists x.A \;\mid\; \forall x.A \;\mid\; \ldots \\
 & \mid & \mathit{Expr}\textbf{:}\mathit{ClassId} \\
 & \mid & \mathcal{M}\!\mathit{ay}\mathcal{A}\!\mathit{ffect}\;\textbf{(}\;\mathit{Expr,Expr}\textbf{)} \\
 & \mid & \mathcal{M}\!\mathit{ay}\mathcal{A}\!\mathit{ccess}\textbf{(}\;\mathit{Expr,Expr}\textbf{)} \\
 & \mid & \mathit{Expr}\;\textbf{\textit{obeys}}\;\mathit{SpcId} \\
 & & \\
\mathit{PredDescr} & ::= & \textbf{predicate}\; R\textbf{(}\;\mathit{ParId}^*\;\textbf{)}\;\textbf{\{}\;\; A\;\;\textbf{\}}
\end{array}
$$

Two-state assertions Two-state assertions allow us to compare properties of two different states, and thus say, e.g. that $\texttt{c.count}_{post} = \texttt{c.count}_{pre} + 1$. To differentiate between the two states we use the subscripts pre and post.

Definition 11 (Two-state Assertions)**.**

$$
\begin{array}{lcl}
t & ::= & \mathsf{pre} \;\mid\; \mathsf{post} \;\mid\; \epsilon \\
B & ::= & A_t \\
 & \mid & \mathit{Expr}_t \geq \mathit{Expr}_t \;\mid\; \ldots \\
 & \mid & \mathcal{N}\!\mathit{ew}(\mathit{Expr}) \\
 & \mid & B \wedge B \;\mid\; \ldots \\
 & \mid & \exists x.B \;\mid\; \forall x.B\,.
\end{array}
$$

Given the syntax from above, we can express assertions like

$\forall \texttt{c.c} :_{pre} \texttt{Counter}.$

$\texttt{c.owner} =_{pre} \texttt{scd} \rightarrow \texttt{c.count}_{pre} = \texttt{c.count}_{post}$,

to require that the `count` of any `Counter` owned by `scd` is immutable across the to states. Notice that for legibility, we annotate infix predicates rather than whole assertions, *e.g.* we write $\texttt{c.owner}{=_{pre}}\texttt{scd}$ to stand for $(\texttt{c.owner=scd})_{pre}$.

4.2 Policies and Validity

Policies can have one of the three following forms: 1) one-state invariants of the form A, which require that A holds at all visible states of a program; or 2) two-state invariants of the form $A\,\textbf{\{}\;\texttt{code}\;\textbf{\}}\;B$, which require that execution of `code` in any state satisfying A will lead to a state satisfying B wrt the original state or 3) any-code invariants of the form $A\,\textbf{\{}\;\texttt{any\_code}\;\textbf{\}}\;B$ which requires that execution of *any* code in a state satisfying A will lead to a state satisfying B wrt the original state.

Definition 12 (Policies)**.**

$$
\begin{array}{lcl}
\mathit{Policy} & ::= & A \mid A\;\{\mathit{code}\}\;B \mid\; A\;\{\mathit{any\_code}\}\;B \\
\mathit{PolSpec} & ::= & \mathit{spec}\;\mathit{SpcId}\,\textbf{\{}\;\;\mathit{Policy}^*\;\textbf{\}}
\end{array}
$$

Validity of one-state assertions We first define validity of one-state assertions. Let $\sigma = (\phi, \chi)$ be a state. Then write $\sigma[v{\mapsto}\iota]$ as shorthand for $(\phi[v{\mapsto}\iota], \chi)$.

Definition 13 (Validity of one-state assertions)**.** *We define the validity an assertion A:*

$\models\; \subseteq \; Module \times state \times Assertion$
using the notation $M, \sigma \models A$:

- $M, \sigma \models \mathtt{e}$ *iff* $\lfloor \mathtt{e} \rfloor_{M,\sigma} = \textbf{true}$.
- $M, \sigma \models R(\mathtt{e}_1, \ldots \mathtt{e}_n)$ *iff*
  $M, \sigma \models R[\mathtt{e}_1/p_1, \ldots \mathtt{e}_n/p_n]$
  where $M(P) = \textbf{predicate}\; P\;\textbf{(}\;p_1 \ldots p_n\textbf{)}\;\textbf{\{}\;\; A\;\textbf{\}}$,
  undefined, otherwise.
- $M, \sigma \models \mathtt{e}_1 \geq \mathtt{e}_2$ *iff* $\lfloor \mathtt{e}_1 \rfloor_{M,\sigma} \geq \lfloor \mathtt{e}_2 \rfloor_{M,\sigma}$.
- $M, \sigma \models A_1 \wedge A_2$ *iff* $M, \sigma \models A_1$ *and* $M, \sigma \models A_2$.
- $M, \sigma \models \exists x.A$ *iff for some address* ι *and some fresh variable* $z \in \mathit{VarId}$*, we have* $M, \sigma[z \mapsto \iota] \models A[z/x]$
- $M, \sigma \models \forall x.A$ *iff for all addresses* $\iota \in \mathit{dom}(\sigma)$*, and fresh variable* z*, we have* $M, \sigma[z \mapsto \iota] \models A[z/x]$.
- $M, \sigma \models \mathtt{e}{:}\mathtt{C}$ *iff* $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \downarrow_1 = \mathtt{C}$.
- $M, \sigma \models \mathcal{M}\!\mathit{ay}\mathcal{A}\!\mathit{ffect}\textbf{(}\;\mathtt{e}, \mathtt{e}'\textbf{)}$ *iff there exists method* $\mathtt{m}$*, arguments* $\bar{\mathtt{a}}$*, state* σ'*, identifier* z*, such that* $M, \sigma[z \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}], z\,.\,\mathtt{m}(\bar{\mathtt{a}}) \leadsto \chi'$*, and* $\lfloor \mathtt{e}' \rfloor_{M,\sigma} \neq \lfloor \mathtt{e}' \rfloor_{M,\sigma\downarrow_1,\chi'}$.
- $M, \sigma \models \mathcal{M}\!\mathit{ay}\mathcal{A}\!\mathit{ccess}(\mathtt{e}, \mathtt{e}')$ *iff there exist fields* $\mathtt{f}_1$*,...* $\mathtt{f}_n$*, such that* $\lfloor z.\mathtt{f}_1 \ldots \mathtt{f}_n \rfloor_{M,\sigma[z \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}]} = \lfloor \mathtt{e}' \rfloor_{M,\sigma}$.
- $M, \sigma \models \mathtt{e}\;\textbf{\textit{obeys}}\;\mathit{PolSpecId}$ *iff*

  $\forall\,(\sigma, \mathtt{stmts}) \in \mathcal{A}\mathit{rising}(M).\; \forall i {\in} \{1..n\}.$

  $\forall\,\sigma', \mathtt{stmts}'.\; (\sigma', \mathtt{stmts}') \in \mathcal{R}\mathit{each}(M, \sigma, \mathtt{stmts}).$

  $M, \sigma'[z \mapsto \lfloor e \rfloor_\sigma] \models \mathit{Policy}_i[z/\textbf{this}]$

  where z *is a fresh variable in* σ'*, and where we assume that PolSpecId was defined as*
  $\texttt{specification}\;\mathit{PolSpecId}\;\{\;\mathit{Policy}_1, \ldots \mathit{Policy}_n\;\}$,

Validity of two-state assertions Validity of two-state assertions $M, \sigma, \sigma' \models B$ is defined similarly to one-state assertions, using cases:

Definition 14 (Validity of Two-state assertions)**.** *We define the judgment*

$\models\;\; \subseteq \;\; Module \times state \times state \times TwoStateAssertion$
using the notation $M, \sigma, \sigma' \models B$ *as follows*

- $M, \sigma, \sigma' \models A_t$ *iff* $M, \sigma'' \models A$,
  where $\sigma'' = \sigma$ *if t=*pre*, and* $\sigma'' = \sigma'$ *otherwise.*
- $M, \sigma, \sigma' \models \mathtt{e}_t \geq \mathtt{e}'_{t'}$*, iff* $\lfloor \mathtt{e} \rfloor_{M,\sigma_1} \geq \lfloor \mathtt{e}' \rfloor_{M,\sigma_2}$,
  where $\sigma_1 = \sigma$ *if t=*pre*, and* $\sigma_1 = \sigma'$ *otherwise,*
  and $\sigma_2 = \sigma$ *if t'=*pre*, and* $\sigma_2 = \sigma'$ *otherwise.*
- $M, \sigma, \sigma' \models \mathcal{N}\!\mathit{ew}(\mathtt{e})$ *iff* $\lfloor \mathtt{e} \rfloor_{M,\sigma'} \in \mathit{dom}(\sigma') \setminus \mathit{dom}(\sigma)$
- $M, \sigma, \sigma' \models B_1 \wedge B_2$ *iff*
  $M, \sigma, \sigma' \models B_1$ *and* $M, \sigma, \sigma' \models B_2$.
- $M, \sigma, \sigma' \models \exists x.B$ *iff for some address* ι *and fresh variable* z*, we have* $M, \sigma[z \mapsto \iota], \sigma'[z \mapsto \iota] \models B[z/x]$.
- $M, \sigma, \sigma' \models \forall x.B$ *iff* $M, \sigma[z \mapsto \iota], \sigma'[z \mapsto \iota] \models B[z/x]$ *holds for all addresses* $\iota \in \mathit{dom}(\sigma)$*, and fresh variable* z.

For example, for states σ_1, σ_2 where $\lfloor \texttt{x.count} \rfloor_{\sigma_1} = 4$ and $\lfloor \texttt{x.count} \rfloor_{\sigma_2} = 14$, we have
$M, \sigma_1, \sigma_2 \models \texttt{x.count}_{post} = \texttt{x.count}_{pre} + 1$. And we also have that $M, \sigma, \sigma' \not\models \texttt{x.count}_{pre} \geq 1$, and $M, \sigma_1, \sigma_2 \models \texttt{x.count}_{post} \geq 1$.

Adherence to Policies We can now define adherence to policies, $M, \sigma \models_{pol}$ *Policy*: which ensures that the requirements of *Policy* are satisfied in any context arising from M.

Definition 15 (Adherence to Policies)**.**

- $M, \sigma \models_{pol} A$ *iff* $M, \sigma \models A$

- $M, \sigma \models_{pol} A \{\texttt{code}\} B$ *iff*
  $(M, \sigma \models A \;\wedge\; M, \sigma, \texttt{code} \rightsquigarrow \sigma' \longrightarrow M, \sigma, \sigma' \models B)$
- $M, \sigma \models_{pol} A \{\texttt{any\_code}\} B$ *iff*
  $\forall \texttt{code}.\ ((\sigma, \texttt{code}) \in \mathcal{A}rising(M) \;\wedge\; M, \sigma \models A \;\wedge\; M, \sigma, \texttt{code} \rightsquigarrow \sigma' \longrightarrow M, \sigma, \sigma' \models B)$

In order to model open systems, it is crucial that we require that after linking *any* module with the module at hand, the policy will be satisfied. For example, to express that M_5 satisfies `ValidSubject` we need to allow any possible implementation of `ValidObserver` as well as any other code to be linked, and still ensure that the Escrow policies are satisfied.

Definition 16 (Classes adhering to Specifications)**.**

- $M \models_{pol}$ *ClassId* ***obeys*** *PolSpecId* *iff*
  $\forall M', \sigma. (\sigma, \_) \in \mathcal{A}rising(M * M').$
  $M, \sigma \models_{pol} o : \textit{ClassId} \rightarrow o\, \textbf{\textit{obeys}}\, \textit{PolSpecId}$

4.3 Modules

Modules, M, are mappings from predicate identifiers to *Chainmail* assertions as described (and also from identifiers to definitions in then underlying model programming language.) We require implicitly for any module M that $M(P) \in \textit{PredDescr}$ or undefined.

Definition 1 (Modules)**.**

$$\begin{aligned}
\textit{Module} &= \textit{ClassId} \longrightarrow \textit{ClassDescr}\\
\textit{Specification} &= (\ \textit{FunId} \cup \textit{PredId} \cup \textit{SpecId}\ ) \longrightarrow\\
&\quad (\textit{FuncDescr} \cup \textit{PredDescr} \cup \textit{Specification}\ )
\end{aligned}$$

The linking operator $*$ combines module definitions, provided that the modules' mappings have separate domains, and performs no other checks. This reflects the open world setting, where objects of different provenance interoperate without a central authority. For example, taking M_a as a module implementing a counter, and M_b as another module using the counter, M_a and $M_a * M_b$ will be well defined, but $M_a * M_a$ is not. We formally define linking $M * M'$, to be the the union of modules' respective mappings, provided that the domains of the two modules are disjoint:

Definition 2 (Linking and Lookup)**.** *Linking of modules M and M' is*

$$* \ :\ Module \times Module \longrightarrow Module$$

$$M * M' = \begin{cases} M *_{aux} M', & \textit{if } dom(M) \cap dom(M') = \emptyset \\ \perp & \textit{otherwise.} \end{cases}$$

$$(M *_{aux} M')(c) = \begin{cases} M(id), & \textit{if } M(id) \textit{ is defined} \\ M'(id) & \textit{otherwise.} \end{cases}$$

5. Related Work

Behavioural Specification Languages Hatcliff et al. [18] provide an excellent survey of contemporary specification approaches. With a lineage back to Hoare logic [19], Meyer's Design by Contract [34] was the first popular attempt to bring verification techniques to object-oriented programs as a "whole cloth" language design in Eiffel. Several more recent specification languages are now making their way into practical and educational use, including JML [26], Spec♯ [1], Dafny [27] and Whiley [44]. Our approach builds upon these fundamentals, particularly Leino & Shulte's formulation of two-state invariants [28], and Summers and Drossopoulou's Considerate Reasoning [49].

Of particular relevance are recent aspect-oriented specification languages such as AspectJML [47] and AspectLTL [32]. AspectJML is an aspect-oriented extension to JML, in much the same way that AspectJ is an aspect-oriented extension to Java [25]. AspectJML offers AspectJ-style pointcuts that allow the definition of crosscutting specifications, such as shared pre- or post-conditions for a range of method calls. These crosscutting specifications can be checked dynamically along with traditional object-oriented JML assertions. In contrast, *Chainmail* specifications naturally crosscut implementation and specification modules without any special notation, although, lacking wildcards, *Chainmail* is not as flexible as AspectJML. To our knowledge, the semantics of AspectJML have yet to be defined formally, although earlier work by Molderez and Janssens describes the formal core of a similar language [39].

AspectLTL [32] is a specification language based on Linear Temporal Logic (LTL). AspectLTL adds cross-cutting aspects to more traditional LTL module specifications: these aspects can further constrain specifications in modules. In that sense, AspectLTL and *Chainmail* both use similar implicit join point models, rather than importing AspectJ style explicit pointcuts as in AspectJML. AspectLTL has a formal definition, as does *Chainmail*; unlike *Chainmail*, AspectLTL has support for automated reasoning with an efficient synthesis algorithm

In general, these all approaches assume a closed system, where modules can be trusted to coöperate. In this paper we aim to illustrate the kinds of techniques required in an open system where modules' invariants must be protected irrespective of the behaviour of the rest of the system.

Defensive Consistency Defensive Consistency was informally introduced by Miller [35] as part of his work on object capabilities. To our knowledge, Murray's is the only prior attempt to formalise defensive consistency and correctness [41]. Murray's model was rooted in counterfactual causation [30]: an object is defensively consistent when the addition of untrustworthy clients cannot cause well-behaved clients to be given incorrect service. Unlike ours, his was formalised very abstractly, over models of (concurrent) object-capability systems in the process algebra CSP [20], without a specification language for describing effects, such as what it means for an object to provide incorrect service. Both Miller and Murray's definitions are intensional, describing what it means for an object to be defensively consistent. In contrast, *Chainmail* is meant for describing and reasoning about real code, and we provide an expressive, extensional framework for evaluating defensive consistency in actual open systems.

Object Capabilities and Sandboxes. *Capabilities* as a means to support the development of concurrent and distributed system were developed in the 60's by Dennis and Van Horn [6], and were adapted to the programming languages setting in the 70's [40]. *Object capabilities* were first introduced [35] in the early 2000s, and many recent studies manage or verify safety or correctness of object capability programs. Google's Caja [38] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: Maffeis et al. [31] develop a model of JavaScript, demonstrate that it obeys two principles of object capability systems and show how untrusted applications can be prevented from interfering with the rest of the system.

JavaScript analyses. More practically, Karim et al. apply static analysis on Mozilla's JavaScript Jetpack extension framework [24], including pointer analyses. Bhargavan et al. [3] extend language-based sandboxing techniques to support defensive components that can execute successfully in otherwise untrusted environments. Politz et al. [45] use a JavaScript type checker to check properties such as *"multiple widgets on the same page cannot communicate."* Lerner et al. extend this system to ensure browser extensions observe *"private mode"* browsing conventions, such as that *"no private browsing history retained"* [29]. Dimoulas et al. [9] generalise the language and type checker based approach to enforce explicit policies, that describe which components may access, or may influence the use of, particular capabilities. Alternatively, Taly et al. [51] model JavaScript APIs in Datalog, and then carry out a Datalog search for an "attacker" from the set of all valid API calls.

Verification of Dynamic Languages A few formal verification frameworks address JavaScript's highly dynamic, prototype-based semantics. Gardner et al. [16] developed a formalisation of JavaScript based on separation logic and verified examples. Xiong and Qin et al. [46, 55] worked on similar lines. Swamy et al. [50] recently developed a mechanised verification technique for JavaScript based on the Dijkstra Monad in the F* programming language. Finally, Jang et al. [23] developed a machine-checked proof of five important properties of a web browser — again similar to our `any_code` invariants — such as *"cookies may not be shared across domains"* by writing the minimal kernel of the browser in Haskell.

Verification of Object Capability Programs Drossopoulou and Noble [10, 43] have analysed Miller's Mint and Purse example [35] by expressing it in Joe-E and in Grace [43], and discussed the six capability policies as proposed in [35]. In [12], they sketched a complex specification language, and used it to fully specify the six policies from [35]; their formalisation showed that several possible interpretations were possible. They also uncovered the need for another four policies and formalised them as well, showing how different implementations of the underlying Mint and Purse systems coexist with different policies [11]. Most recently, [15] they have shown how the combination of *Chainmail*, an untyped featherweight core capability language $\mathcal{F}ocal$, and novel Hoare logic, can verify a trust-sensitive example (the escrow exchange) in an open world. In contrast, this work focuses on the *Chainmail* specification language and how it can be used to support defensive consistency.

6. Conclusions and Further Work

In this paper we addressed the question of how to specify open systems in an open world. To answer this questions, we contribute:

- The *Chainmail* specification language.
- *Defensively consistent* policies specified in *Chainmail*.
- A *formal model* of the semantics of *Chainmail*.

In further work we will extend our approach to deal with concurrency, distribution, exceptions, networking, aliasing, and encapsulation. We will also consider multi-dimensional modularity constructs in implementations, as well as specifications. Finally, we hope to develop dynamic monitoring and automated reasoning techniques to make these kinds of specifications practically useful.

Acknowledgments

We gratefully thank the Programming Language group at VUW, the SLURP group at Imperial College, the PLAS reviewers, the IFIP WG2.3 and WG2.16 members for valuable discussions. This work is supported in part by the Royal Society of New Zealand Marsden Fund, a James Cook Fellowship, and the EU project Upscale.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, 2005.

[2] BBC: On This Day. 1984: Tory cabinet in Brighton bomb blast, 2015. [Online; accessed 15-October-2015].

[3] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.

[4] D. Clarke, J. Noble, and T. Wrigstad, editors. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*. Springer, 2013.

[5] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*. ACM, 1998.

[6] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. *Comm. ACM*, 9(3), 1966.

[7] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical Report SRC-RR-98-156, Compaq Systems Research Center, July 1998.

[8] E. W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[9] C. Dimoulas, S. Moore, A. Askarov, and S. Chong. Declarative policies for capability control. In *Computer Security Foundations Symposium*, 2014.

[10] S. Drossopoulou and J. Noble. The need for capability policies. In *FTfJP*, 2013.

[11] S. Drossopoulou and J. Noble. How to break the bank: Semantics of capability policies. In *iFM*, 2014.

[12] S. Drossopoulou and J. Noble. Towards capability policy specification and verification, May 2014. `ecs.victoria-.ac.nz/Main/TechnicalReportSeries`.

[13] S. Drossopoulou, J. Noble, and M. S. Miller. Swapsies on the Internet. In *PLAS*, 2015.

[14] S. Drossopoulou, J. Noble, M. S. Miller, and T. Murray. More Reasoning about Risk and Trust in an Open World. Technical Report ECSTR-15-08, VUW, 2015.

[15] S. Drossopoulou, J. Noble, M. S. Miller, and T. Murray. Reasoning about Risk and Trust in an Open World, 2015. Under Review.

[16] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *POPL*, 2012.

[17] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA*, 2002.

[18] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. J. Parkinson. Behavioral interface specification languages. *ACM Comput.Surv.*, 44(3):16, 2012.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12:576–580, 1969.

[20] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[21] C. A. R. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: a manifesto. *ACM Comp. Surv.*, 41(4), 2009.

[22] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2), Apr. 1992.

[23] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, 2012.

[24] R. Karim, M. Dhawan, V. Ganapathy, and C.-C. Shan. An Analysis of the Mozilla Jetpack Extension Framework. In *ECOOP*, Springer, 2012.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[26] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Iowa State Univ. `www.jmlspecs.org`, 2007.

[27] K. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[28] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *ESOP*, 2007.

[29] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *ESORICS*, Sept. 2013.

[30] D. Lewis. Causation. *Journal of Philosophy*, 70(17), 1973.

[31] S. Maffeis, J. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE S&P*, 2010.

[32] S. Maoz and Y. Sa'ar. AspectLTL: An aspect language for LTL specifications. In *MODULARITY*, 2011.

[33] R. Merrill. focal: new conversational language. DEC, 1969. `homepage.cs.uiowa.edu/˜jones/pdp8/focal/-focal69.html`.

[34] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[35] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.

[36] M. S. Miller, T. V. Cutsem, and B. Tulloh. Distributed electronic rights in JavaScript. In *ESOP*, 2013.

[37] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments: From object to capabilities. In *Financial Cryptography*. Springer, 2000.

[38] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Safe active content in sanitized JavaScript. `code.google.com/p/google-caja/`.

[39] T. Molderez and D. Janssens. Design by contract for aspects, by aspects. In *FOAL*, 2012.

[40] J. H. Morris Jr. Protection in programming languages. *CACM*, 16(1), 1973.

[41] T. Murray. *Analysing the Security Properties of Object-Capability Patterns*. D.Phil. thesis, University of Oxford, 2010.

[42] J. Noble and R. Biddle. Notes on postmodern programming. Technical Report CS-TR-02-9, VUW, 2002.

[43] J. Noble and S. Drossopoulou. Rationally reconstructing the escrow example. In *FTfJP*, 2014.

[44] D. Pearce and L. Groves. Designing a verifying compiler: Lessons learned from developing Whiley. *Sci. Comput. Prog.*, 2015.

[45] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. Adsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[46] S. Qin, A. Chawdhary, W. Xiong, M. Munro, Z. Qiu, and H. Zhu. Towards an axiomatic verification system for JavaScript. In *TASE*, pages 133–141, 2011.

[47] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. M. F. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm. AspectJML: Modular specification and runtime checking for crosscutting contracts. In *MODULARITY*, 2014.

[48] J. H. Saltzer. Protection and the control of information sharing in Multics. *CACM*, 17(7):p.389ff, 1974.

[49] A. J. Summers and S. Drossopoulou. Considerate Reasoning and the Composite Pattern. In *VMCAI*, 2010.

[50] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, pages 387–398, 2013.

[51] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *SOSP*, 2011.

[52] B. Venners. Failure, preconditions, and reuse: A conversation with Bertrand Meyer, Part IV. `http://www.artima.com/intv/exceptP.html`, Mar. 2004.

[53] Wikipedia. Anna Karenina Principle — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 15-October-2015].

[54] A. Wills. Reasoning about aliasing. In *ECCOP*, 1993.

[55] W. Xiong. *Verification and Validation of JavaScript*. PhD thesis, 2013, Durham University.