Implementing Design Rules in Aesop n+1

Bob Monroe and Ralph Melton
March 27, 1996

This document describes a first-cut at implementing a design rule specification and maintenance system for the new version of Aesop. It is rough, full of holes, and intended primarily to invite discussion over a concrete proposal.

1 Overview

This paper presents a proposal for the design and implementation of a design rule specification and enforcement mechanism that will "live" inside of the Fable Abstract Machine (FAM) in the next generation Aesop system. We begin by listing some of the goals of the system, the assumptions we make, and desired attributes of the final system. We then explicate the various types of policies that need to be supported, present an object model for the design rule structure, give a few of examples that show a range of design rule representations, and close with a discussion of some other issues, including how this model fits with the rest of Aesop.

2 Goals, assumptions, and desired Attributes.

Goals. The design rule infrastructure will need to support the following goals:

1) This schema should support the representation of a sufficiently broad set of design rules, violation policies, and detection policies and mechanisms.
2) The design rule specification should be as modular, declarative, and independent of the underlying Aesop implementation as possible.
3) There should be a hierarchical structure to the rules so that a style designer can specify that there are some rules that are inviolate and others that a user or environment designer can adjust to meet their needs or tastes.

Assumptions. We have made the following assumptions in specifying the design:

1) The cost of evaluating a predicate varies greatly. A user should not be trapped by having an expensive predicate evaluated unnecessarily.
2) The style designer decides whether a rule is an invariant. If it is not specified as an invariant, then an environment designer may adjust and possibly weaken the enforcement and detection policies for the rule. It's not clear whether a tool designer may do the same.
3) Every design rule can be checked explicitely by an external tool through some sort of request mechanism.
4) Some design rules may be checked implicitely, as a result of operations on a design.
5) Any modification to a design may involve implicit checks.

Desired attributes (for implicit checks):

1) A decoupling of the checks for rules that apply at the same time (i.e. dependencies amongst rules).
2) Dynamic bindings of implicitely checked rules to instances
3) Support for both pre-checks and post-checks.

3 Policy classification and specification.

We have come up with three classes of policies to use when dealing with design rules: detection policy, enforcement policy, and an invariant flag. Detection policies determine when the FAM checks a design rule's predicate to determine that it hasn't been violated. The range of detection policies have now been reduced to two:

- test that the rule holds when explicitely requested to do so (i.e. via a request from outside the FAM)
- test the rule whenever something was done to the design that could change the value of the test.

It is not yet clear to me who sets the detection policies and whether they may be changed by requests from outside the FAM (i.e. environment requests or tool requests to change them).

Enforcement (or violation) policies determine what the FAM should do when it detects a design rule violation. It is not clear whether the enforcement rules apply only to implicit detection of design rule violations or explicit detections as well. There is, I think, a reasonably strong argument to be made that the tool that requested a design rule be checked should set it's own policy for dealing with the violation. An enforcement policy of "forbid," for example, doesn't make much sense if the design rule is evaluated at the request of an external tool that simply wants a response. In either case, we have specified the following four enforcement policies:

- forbid
- warn and confirm
- warn and continue (no confirmation)
- ignore

Finally, we've added a notion of an "invariant" flag. The invariant flag allows a style developer to specify that a design rule is fundamental to the style and can not be violated under any circumstances. Thus neither an environment designer nor a tool builder can weaken an "invariant" design rule. The need for the invariant flag arose from a recognition that we should allow the specification of both very strict and fundamental design rules, which are now labelled invariant, as well as design rules that are more advisory, or rules of thumb (not an invariant). An example of an invariant design rule would be that only pipe and filter connectors and components can be added to a configuration in a pipe-filter style. A design rule that simply suggests a topology for optimal throughput, on the other hand, is not an invariant, and the user may well want to take part in the decision to follow or ignore the suggestion. The invariant flag suggests appropriate detection and enforcement policies (test always and forbid), but it is in some ways orthogonal to the other policy mechanisms themselves. The details of the invariant flag have not yet been worked out.

4 Design rule object model

Figures 1-3 contain OMT-ish diagram that depict the basic design rule class, along with the relationships between the design rule class, instances of the class, and the design element instances that are attached to the design rule instances.

Clearly, the examples in figures 1-3 require some explanation. Please do not be overly concerned with the mishmash of notations. We started with an OMT-ish object model and added other notations where and as they seemed appropriate. This design representation is intended primarily to expose issues and questions. In order to make the figures more comprehensible, I will describe the object fields in detail here:

The first list contains the items that are static across all instances of the design rule class.

- name : a unique identifier for the design rule class
- signature : the signature describes the types of entities over which the design rule can be checked.
- test : test() is the method made available to tools outside of the FAM for requests to evaluate whether the design rule holds for a given set of design elements with a given state. The signature of the test function is specified in the "signature" field.
- methods : this list of base methods -- methods that are available on the component, connector, port, role, and aggregate base classes -- specifies all of the base methods that could cause the design rule to be violated. Although there can be derived methods that change the design, they need to do so through the methods provided by the base classes for this scheme to work. [side note - there's an issue here about how we extend classes]
- pre-checks : specifies a mapping from the methods listed in the previous field, to functions that perform the prechecks on these methods to make sure that the method will not violate the design rule. There has to be at least one entry for each of the methods listed above, though the function can simply return "no problem" if there is an appropriate post-check to do the check instead, or it can somehow be proven by the style designer that the check will always return true. It is acceptable for a single method to have multiple pre-check function calls, but it must be noted that the signature provided to each pre-check for a given method must be the same across all design rules. For example, whenever a pre-check is called for "configuartion::attach(p,r)" the pre-check method must accept the arguments (p : port, r:role).
- post-checks : are like pre-checks, except that they occur after the base method has been called. Post-check functions have the same signature as test(), which they call to detect a violation. Unlike pre-checks, post-checks also need to provide a violation policy for each check to specify how to deal with a detected violation.
- Invariant : boolean that specifies whether the design rule is an invariant.

In addition to the static elements, each instance of a design rule can store local, instance specific data; for caching checks, for example. Each instance also specifies its enforcement policy, which may differ amongst various instances of the same rule. In general, the instance-specific data will be specified by subclassing the design rules and adding the requisite data.

5 Other issues

Although there are many open issues relating to this design rule schema, I will discuss two significant issues here that we at least partly address – how does the actual pre and post-check code get distributed and invoked, and whether this scheme works for both complete styles and individual instances.

5.1 Distribution and invocation of checking code

The schema described previously does not fully support the highly declarative design rules that we would like to eventually be able to support. It is still up to the designer to write the code that performs the checks and figure where the checks need to be performed. The style designer has a proof obligation that says he or she must show that the design rule is indeed checked at any method call in which it can be violated. We can, however, provide some structure to assist the style designer in writing simple, modular checks and simply specifying the appropriate methods for which they should be invoked. The basic model that we are proposing is that a style designer specifies the base methods that can cause a violation, along with an appropriate pre-check function to be run as a check before executing that method. This specification format is depicted in the object diagrams of figures 1-3. What we were previously describing as "detection policy" is now specified implicitly in the specification of the functions to perform the pre-checks and the methods for which they must be performed.

In order to actually perform the checks, each design element will keep a list of references to design rules (instances) by which it needs to abide. Whenever a base method is called on a design element, the method will begin by requesting all of the relevant checks for the self element. This request will return a list of the checks for all of the relevent design rules that can be affected by this operation. This list is a projection of the whole set of design rule checks attached to this design element, filtered by the particular base method being invoked. Before performing the base method itself, all of the relevent checks are performed. If none of them raise a violation then the method itself is run. If, on the other hand, one of the checks does raise a violation, the enforcement policy for the rule that raised the violation is retrieved, and handled accordingly. After the method is invoked there is a corresponding process for dealing with post-checks.

Realizing that this description is probably rather confusing, please look at the C++ish code shell of a base method below:

```
component::add_port(p)  {
  // perform the pre-checks
  foreach r in self.rules() {
    pre_ck = r->get_pre_checks for component::add_port
```

```
    if (pre_ck)  {
      v = r->*pre_ck(this, p);  // call the rules precheck method
       handle_violation(v);
    }
  }

  // ... perform add_port method

  // then perform the post-checks
  foreach r in self.rules() {
    post_ck = r->get_post_checks for component::add_port
    if (ck)  {
       //call the rules postcheck function.  It's not clear
       //whether this is a unique function or simply test()
       v = r->*post_ck(this, p);  // or r->test(sig)
       handle_violation(v);
    }
  }

} // end method
```

5.2 Style-wide rules vs. instance-level rules.

One of the advantages of using this scheme for specifying design rules is that it appears to be able to work for both instance level design rules as well as style-wide design rules. Design rules can be attached to specific instances of design elements in a straightforward way, because the scope of a particular design rule instance is never greater than a (singleton?) set of design elements. Fortunately, it is seems much easier to add the notion of style-wide design rules to a scheme that supports instance level design rules than vice-versa. That appears to be true here as well. In order to make a design rule apply to an entire style instead of a single element, an appropriate command need simply be added to the constructor for the types that will be used in that environment. A snippet of code that might implement this appears below:

```
class Filter : Component {
   Filter() {
     add_design_rule(Filterness);
     ... rest of constructor
   }
...
}
```

Although this still has the undesirable effect of keeping the type definitions tightly intertwined with the style definitions at an implementation level, this seems to be a simple and regular enough addition to the constructors that we can probably provide a pre-processing tool (ie the style designer's workbench) that will do this linking for us. By providing a small set of modular pieces that are composed to form a style the items may be tightly tied together in the "compiled" or at least interpreted code, but the constiuent elements can still be readily interchanged and reused without running into serious dependencies. This hypothesis might not, of course, work out very well in practice.

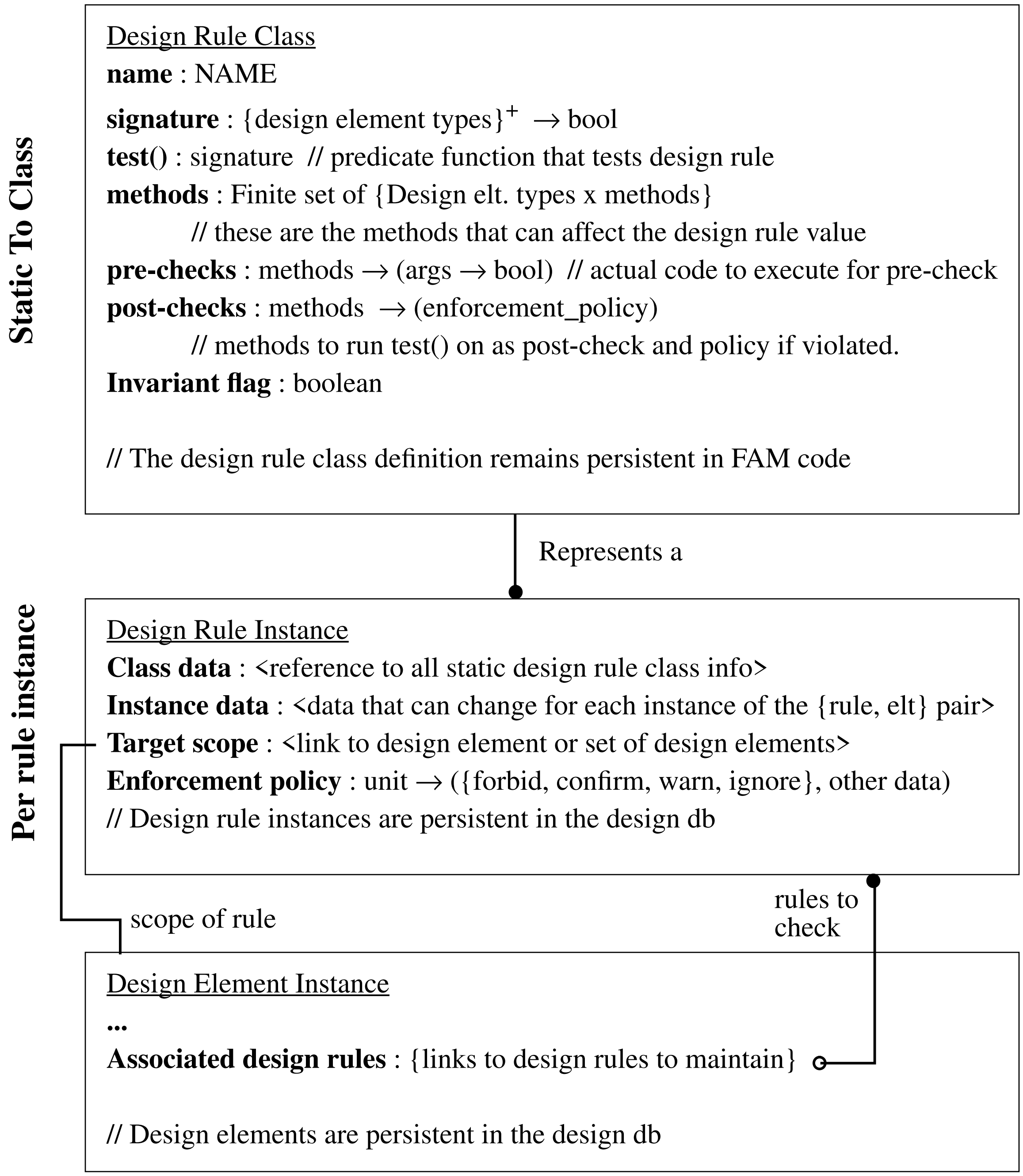

Figure 1. Design rule object model

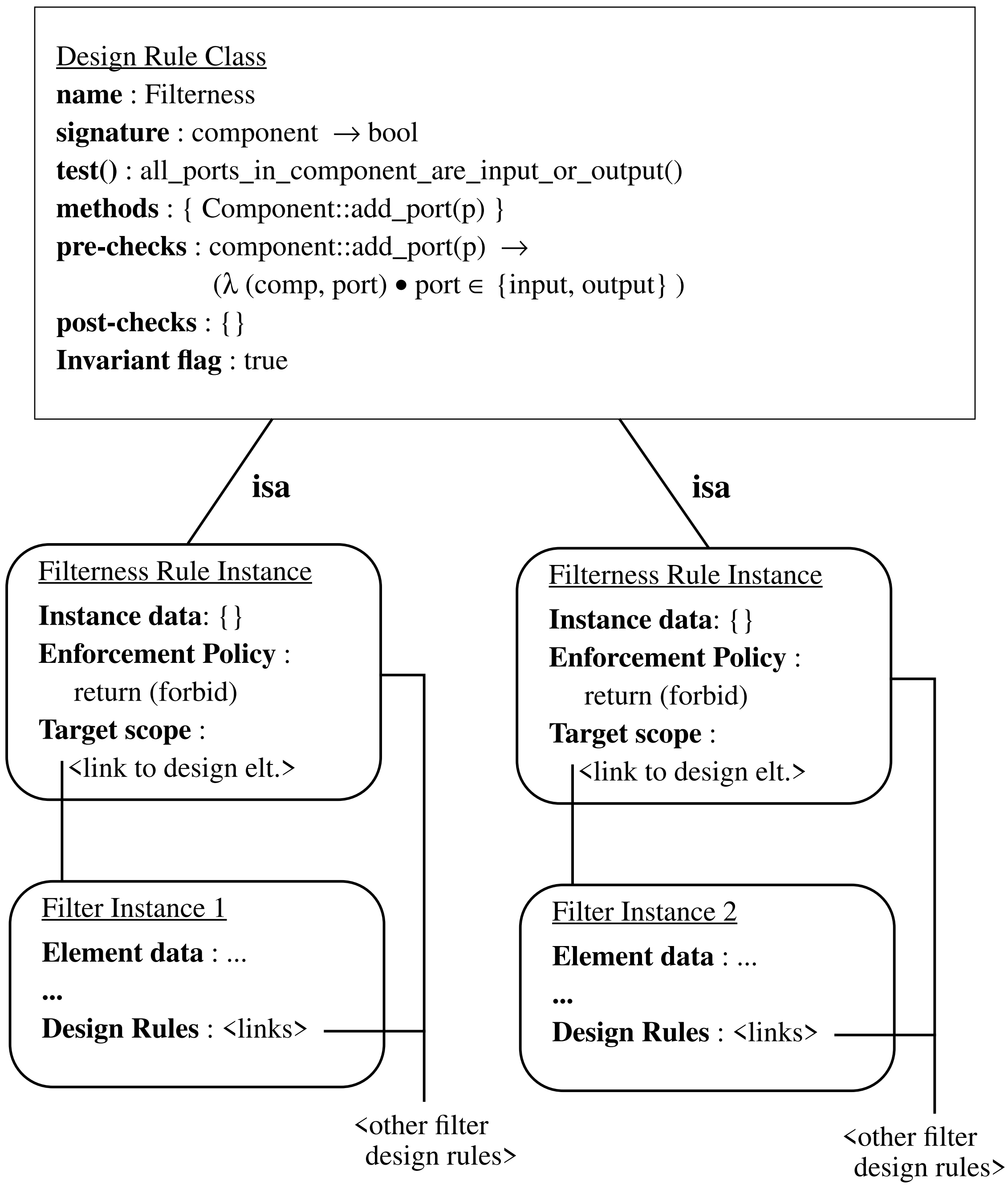

Figure 2. Example of a "Filterness" design rule

Design Rule Class
name : Valid_pf_attachments
signature : config $\rightarrow$ bool
test() : (forall attached(port, role) in config •
(port, role) $\in$ {(input, sink), (output, source)})
methods : { configuration::attach(p,r) }
pre-checks : configuration::attach(p,r) $\rightarrow$
(λ (config, port, role) • (port, role) = (input, sink)
$\vee$ (port, role) = (output, source))
post-checks : { }
Invariant flag : true

Instance Data : { }
Enforcement Policy : forbid

Design Rule Class
name : cycle_free
signature : configuration $\rightarrow$ bool
test() : no_cycles_in_configuration(configuration)
methods : { configuration::attach(p,r), configuration::detach(p,r) }
pre-checks : configuration::attach(p,r) $\rightarrow$
(λ (config, port, role) • creates_a_cycle(config, port, role))
configuration::detach(p,r) $\rightarrow$
(λ (config, port, role) • removes_a_cycle(config, port, role))
post-checks : configuration::attach(p,r) $\rightarrow$ test()
Invariant flag : false

Instance Data : current_state : transitive_closure_matrix;
previously_warned : bool;
Enforcement Policy : (λ () • if (!previously_warned)
then (previously_warned = true; return (warn, “cycle detected”))
else (return ignore))

Figure 3. Examples of two more design rules. In this case the notation has changed to save space. Instance specific attributes are now listed in the bottom box of each rule.