Defining a Security Reference Architecture †

Sigurd Meldal   *David C. Luckham*

Technical Report CSL–97–728
Program Analysis and Verification Group Report No. 76

June 1997

Computer Systems Laboratory
Department of Electrical Egineering
Stanford University
Stanford, CA 94305–9040

Abstract
This report discusses the definition and modeling of reference architectures that specify the security aspects of distributed systems. NSA's MISSI (*Multilevel Information System Security Initiative*) security reference architecture is used as an illustrative example. We show how one would *define* such a reference architecture, and how one could use such a definition to *model* as well as *check* implementations for compliance with the reference.

We demonstrate that an ADL should have not only the capability to specify interfaces, connections and operational constraints, but also to specify *how* it is related to other architectures or to implementations.

A reference architecture such as MISSI is defined in *Rapide* [10] as a set of hierarchical interface connection architectures [9]. Each Rapide interface connection architecture is a *reference* architecture – an abstract architecture that allows a number of different implementations, but which enforces common structure and communication rules. The hierarchical reference architecture defines the MISSI policies at different levels – at the level of enclaves communicating through a network, at the level of each enclave being a local area network with firewalls and workstations and at the level of the individual workstations. The reference architecture defines standard components, communication patterns and policies common to MISSI compliant networks of computer systems. A network of computers may be checked for conformance against the reference architecture.

The report also shows how one can generate architecture scenarios of networks of communicating computers. The scenarios are constructed as Rapide executable models, and the behaviors of the models can be checked for conformance with the reference architecture in these scenarios. The executable models demonstrate how the structure and security policies in the reference architecture may apply to networks of computers.

Key Words and Phrases: Software architectures, security, reference architecture, software engineering, specification, testing, conformance.

† This project was funded by TRW under contract 23679HYL6M, DARPA under F30602–95–C–0277 (subcontract C-Q0097), and by NFR under contract 100426/410.

1. Introduction

This report investigates the use of the *Rapide* ADL in the definition of elements of NSA's MISSI reference architecture [7]. Everybody knows what an *architecture* is – it is a set of *components* and *connections* between them. However, that is as far as agreement goes. What the proper methods of defining these entities are, what *conformance* means, what the distinctions are between an *architecture*, and *architecture style* and a *reference architecture*, these are issues that are unresolved (and presumably unresolvable, as they are questions closely related to world-views, methods and consequently often come down to pseudo-religious beliefs).

Architectures are used in different situations, and for distinct reasons. The most concrete use is in designing software systems, to make an initial sketch of it in terms of its module decomposition architecture in the top-down tradition of design, focusing on the high-level components and their means of interaction. Architectures are also used to define *references* against which implementations can be checked for compliance. Such reference architectures define the functional components of the architecture and how the components may interact, but need not require that distinct components in the architecture necessarily be distinct also in the implementation. The use of reference architectures allows a separation of concerns in the system specification – distinct reference architectures address distinct aspects of the system (*e.g.,* there might be one reference architecture stating fault-tolerance requirements, another (such as the MISSI reference) stating security requirements, another (such as the ISO OSI reference stack) addressing communication protocols, etc.).

The presence of a component or connection between components in a reference architecture may signify different requirements, depending on which aspect of the system the reference addresses. *E.g.*, does the lack of a connection between two modules indicate a prohibition against their direct interaction (*i.e.*, is the interaction graph as given by the architecture supposed to be complete)? Does a connection between two components indicate that they *will* communicate (*i.e.* a connection represents not only a potential for interaction, it is also a requirement that such an interaction shall occur)? And in all cases, what is the concept of interaction anyway? Does an architecture imply what protocol an interaction shall adhere to? *E.g.* RPC vs. buffered pipes vs. passive, reactive systems vs. event broadcasting, etc. In the end, what distinguishes one kind of architecture from another is the *conformance requirements* imposed by the architecture.

This report discusses how one can capture a security reference architecture in a manner amenable to analysis and automatic conformance checking. After giving a brief overview of the *Rapide* ADL in section 1.1, in section 2 we present the process of architecting using the *Rapide* ADL, giving examples from the MISSI reference architecture. In section 3 we go through all the top level requirements of the MISSI reference architecture one by one, showing how they are captured in the *Rapide* ADL. In section 4 we shall briefly look at how the reference architecture can be put to use for (semi-) automatic checking, visualization and analysis of implementation system conformance.

1.1 The *Rapide* ADL

In reading an architecture description, the question of what the description actually *means* needs to be resolved unambiguously in the readers' and designers' mind in order to evaluate and then implement a given architecture. Without a clear understanding of the semantics of a notation (be it graphical – boxes and arrows, or textual – as in *Rapide*) one cannot be sure that whatever is extracted from it (be it implementation strategies, modeling results, etc.) is implied by the description given, and understood by other readers of the architectural description.

An *interface connection architecture* [9] is defined by giving its

- *Components*: the primary elements of the architecture, and their means of interaction with other components. Components are considered black boxes constrained only by the definitions of their *interfaces*.
- *Connections*: the lines of interaction between components.
- *Conformance*: identifying minimum requirements of how an implementation may satisfy the architecture.

The ***Rapide*** model of architectures is *event based* – a basic notion being that architecture components are defined by the kinds of events they may generate or react to. An interface also identifies the semantics of a conforming component by giving event based *constraints,* specifying whether particular protocols are to be adhered to, identifying causal relationships between events, etc. Such constraints form the basis for analysis and testing tools, such as run-time checking for conformance violations [6, 17].

A successful ADL requires a high degree of flexibility in how an architecture can be refined. Naturally one wants to be able to refine interface definitions, making use of subtype substitutivity when extending an interface with new capabilities or by adding further constraints. In addition to this basic capability, an ADL should enable the definition of *hierarchies* of architectures, where one architecture can be interpreted quite flexibly as an implementation (or refinement) of another. The ***Rapide*** *map* construct gives the designer the tool to explicitly define how complex patterns of events in one architecture correspond to more abstract events of another, thereby enabling a powerful and checkable notion of *conformance*.

The literature presents a number of distinct ways of distingushing *kinds* of architectures (*e.g.,* Soni et al. [24] makes a distinction between *object* and *function* decomposition architectures, among others). We prefer the notion that "an architecture description conveys a set of *views,* each of which depicts the system by describing domain concerns." [5] The distinction between different architectures descriptions then becomes one of a *difference of conformance requirements*. In moving from (say) a *module decomposition architecture* to an implementation, conformance would require disjoint sets of modules implementing distinct components of the architecture. In contrast, in checking whether a *reference architecture* is satisfied by a particular implementation one would make the weaker conformance requirement that there be a *mapping* of components and events at the implementation level to components and events of the reference architecture.

This perspective on what an architecture is allows a clean separation of concerns. One can specify multiple architectures for any given implementation, each focusing on a particular aspect of the system, each with an appropriate set of conformance requirements. For instance, when specifying a distributed object system it is reasonable to separate *security* concerns from *fault tolerance* concerns. Part of the security architecture for the system would state the conformance requirement that information should flow *only* along connections defined in the architecture; the architecture identifies the *maximal* connectivity of an information flow graph. In contrast, part of the fault tolerance architecture for the system would be to state the conformance requirement that information should be able to flow *independently* along all connections defined in the architecture, making no restrictions on the presence of extra connections; the architecture identifies the *minimal* connectivity of an information flow graph. In claiming that a particular implementation satisfies both perspectives the implementor would explicitly give the two maps, from the implementation to each of the reference architectures, showing the conformance argument.

The vocabulary of the ***Rapide*** ADL [10] incorporates and extends the basic vocabulary of interface connection architectures:

Components: The computational entities of an architecture.

Connections: The means by which components interact. Connections have a limited computational power, invoked when determining where a particular interaction is routed.

Events: Representing that something happened. What that something *is* may vary from architecture to architecture, and with varying degrees of abstraction.

Reactive rules: Representing state-machine-like behaviors, implementing or simulating components.

Patterns: Descriptions of how events may be related by causality, time or other relations. Patterns are described using an extension of regular expressions with placeholders to describe partial orders of events.

Constraints: Predicates, usually in the form of prescribed or proscribed patterns of behavior, indicating the intended functionality of a component.

Maps: Interpreting an implementation as being of a particular architecture – useful for constraint checking, and when relating a model or implementation to a particular reference architecture.

In specifying the MISSI reference architecture there were two features of *Rapide* that were particularly useful:

Causality: In *Rapide* one can specify whether particular patterns of events should be independent or causally related. This allows a very precise description of information flow.

Maps: In *Rapide* one can specify how distinct architectural descriptions are related, and precisely how an implementation satisfies a given specification. This allows multiple views of a system, each with its distinct map showing how conformance is obtained.

Rapide's object-oriented type- and module definition sublanguage provides features for code refinement and reuse (through inheritance and polymorphism) and specification refinement and reuse (through subtyping and polymorphism). The reactive rules of *Rapide* object types provide a limited synthesis capability; when the behavioral constraints of a group of components (such as, say, the "Certificate Authorities") are given in a particular form the tools can synthesize conforming behaviors, supplying an early operational model of the reference architecture.

The *Rapide* execution model, emphasizing causal and temporal relationships between events of a system, provides the capability to be quite specific about how components of an architecture may (or may *not*) interact. Causal relations can often identify whether assumptions about the degrees of independence among an architecture's components are warranted or not. *E.g.*, the focus on causal relationships allows the *Rapide* user to state in very general terms assumptions about the presence of covert channels, and to identify possible means of covert interaction in an architecture through the analysis of causal relationships displayed by test executions.

Furthermore, it allows tools to investigate the causal relations between events, distinguishing between temporal relationships that are causally significant and those that are not.

The *Rapide* pattern and constraint languages supports the definition of operational policies and specific protocols, which can take into account *causal*- as well as *time*-relationships between events.

The *Rapide* map construct supports explicit statements of conformance – the implementor of an architecture can state *exactly* how the implementation conforms: it defines which (sets of) components of the implementation play the role of particular components of the architecture, how patterns of events in the implementation correspond to more abstract events used in the architecture, etc. Since maps are given explicitly, they allow tools to check for conformance automatically, avoiding laborious reasoning or formal proofs except where exceptional circumstances requires an extraordinary degree of confidence in the implementation's conformance.

The map construct is also a valuable tool whenever an architecture is given a *hierarchical structure*. *E.g.,* if one level of structure is defined in terms of federations of *enclaves* connected via *wide area networks*, and another level as network-connected *workstations*, *certificate servers*, etc., then maps are the means whereby the distinct levels can be related in the architecture definition. For instance, through the definition of appropriate maps the designer can identify how the set of networks, workstations and servers aggregate into enclaves and WANs (see section 2.6).

1.2 Secure architectures

There are a number of perspectives one may apply when discussing the security aspects of a software architecture. In particular, in this document we shall address two aspects of the MISSI reference architecture:

Structures: That the secure architecture has a certain structure, requiring the existence of certain components (such as "certificate authorities," or "enclaves" [7]). The structure may be defined at different levels of abstraction, with different conformance requirements. We deal with

1. a *global* level, focusing on the main components and the overall constraints on their interaction. At this level general policies about information flow and the like may be stated, without regard to how these policy constraints are ensured by particular protocols, functional units, etc.
2. *a concept of operations* ("*conops*") level, focusing on the functional decomposition of the architecture, identifying the events of interest, the main functional components and their potential for interaction.
3. an *execution* level, describing the dynamic, physical structure of the system.

The architectures at each of these levels are related to one another and impose different conformance requirements on the implementation. Both the relationships and the conformance requirements must be defined.

Information flow integrity: That certain policies and procedures regarding the authorization and acceptability of information are adhered to as it is being generated and propagated. Such policies may be in terms of any of the three levels listed above and could also involve references to cryptographic and encoding requirements, as well.

2. The Architecting Process

The MISSI reference architecture is defined in a series of prose documents, some with first order predicate logic definitions of MISSI policies. In this exposition we shall stay with the overview document, given in full in [7]. The overview is an executive summary of the reference architecture, but contains enough detail to evaluate the utility of ***Rapide*** to specify the architecture.

We find the *process* of *constraints capture* in itself very useful. This process can be quite enlightening – interpreting the prose and giving it an unambiguous meaning often identifies potential contradictions or holes in the original definitions of the reference architecture. Even in the case where the final reference document is given in prose, we find that the exercise of formalizing the prose as it is being developed may help the development team, by enhancing their understanding of the interplay of their own statements.

Reference documents are also subject to mishaps, resulting from typographical mistakes through incomplete version-control to out-right conceptual misunderstandings. The sheer size of most such documents make them hard to check for consistency and correctness unless such checks are assisted by (semi-)automatic tools. Consequently, the presence of supporting tools should be almost mandatory in the definitions of standards. Tools require the existence of (parts of) the standard in a machine-manipulatable form, *i.e.*, in the form of a formalized set of definitions.

2.1 Prose and Constraints Capture

The process leading up to a formal capture of an architecture has three main steps: (1) identifying the components, (2) identifying how they are connected, and (3) identifying how the connections are used. The three steps are accompanied by a fourth, stating the conformance requirements, when relating the architecture to an implementation (or model, or a more detailed version). We'll go through the process of capturing the MISSI reference overview, giving examples of each of these steps.

Capturing the interface connection architectures defined in the MISSI specification, we first identify the *levels* of the reference architecture. In this report we shall deal with two levels, the *global* and the *concept of operations* levels (see section 1.2 above).

For each level we proceed to identify and define the *components* of the level by defining their interfaces (sections 2.2, 2.5.1), and then going on to define the *connections* among them (sections 2.3, 2.5.3) and how they are used (sections 2.4, 2.5.3)

As appropriate, we then go on to define how the components and activities of one level map to those of another (section 2.6).

2.2 What are the components?

For each kind of component (such as an *enclave* at the global level) we define a ***Rapide*** *type*, whose interface is developed as the architecture is being refined. Part of this definition may identify how one type is a refinement or *subtype* of another [15]. Of course the interface definitions themselves rely on other types (such as *security classifications* and *security tokens*) already having been defined.

A very first approximation of an *enclave* type is given in Figure 1.

It identifies two key characteristics of an enclave:

1. The *provides* declaration of **s_class** makes it possible to refer to the security attributes (here exemplified by it having a security classification) of every enclave.
2. The *service* declaration of **wan_conn** states that every enclave interface contains a **Flow** entity which (as we shall see) defines the minimum communication capabilities of enclaves.

Architecture component interfaces can be highly structured. It may be helpful to think in terms of *plugs* and *sockets* [9]: a component's interface offers a set of distinguishable means of connecting it to its environment, similarly to what one expects in the hardware world. Such a means of connecting come in dual forms (as in *plugs* and *sockets* being duals in hardware), and may have further substructures (as in a single plug carrying pins/sockets for a number of wires).

```
type Enclave is interface
provides
        s_class     : SecurityClassification;
service
        wan_conn   : Flow;
end Enclave;
```

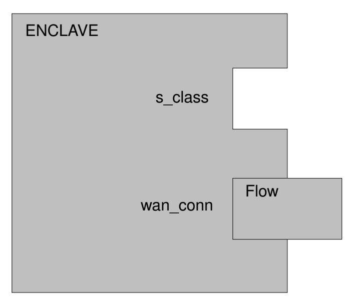


Figure 1: A definition of an enclave type

It is natural to depict the **Flow** service type graphically (Figure 2), similarly to how we depict the **Enclave** interface definition in Figure 1. We can see that the **wan_conn** attribute has a structure; the declaration of its type, **Flow**, shows that **wan_conn** consists of two **action** declarations. An *out* action declaration indicates that the component may generate events which its environment may observe, an *in* action declaration indicates that the component may react to events generated by the environment. The **wan_conn** declaration is therefore in fact a bi-directional communication interface offering both a means of sending messages to the environment (intended to be a WAN) as well as of accepting such messages from the environment.

In ***Rapide***, such structured communication interfaces are called *services*. The dual of the **wan_conn** service will be part of the interface of the wide area network component of the architecture, and is naturally depicted as the inverse of the Flow type (*i.e.*, it forms a plug to the Flows socket). Where the type Flow has an *out action* there will be a corresponding *in action* of the dual, and vice versa. One need not declare dual types explicitly, but can instead use the keyword **dual**. We have given the dual of Flow explicitly in Figure 2.

```
type Flow is interface
action
    out    Release   (data : Data; destination : Address);
    in     Accept    (data : Data; destination : Address);
end Flow;

type DualFlow is interface
action
    in     Release   (data : Data; destination : Address);
    out    Accept    (data : Data; destination : Address);
end Flow;
```

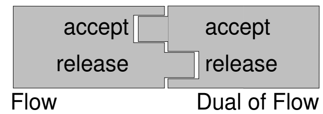


Figure 2: Plugs and sockets

Though plausible as a first approximation in the global view of a distributed system, we may want to add some instrumentation points to the definition of an enclave. Consequently, in Figure 3 we create a subtype of the Enclave type[1]. We introduce a new out action called internal to be able to speak about things going on within the enclave (leaving the notion of "Activity" uninterpreted for now). As we shall see later (page 8), this turns out to allow an interesting architectural constraint about the existence of covert channels.

```
type MISSI_Enclave is
    include Enclave;        -- Create a subtype of the Enclave type
interface
provides
    function   release_reviewers ()              --   (See page 16.)
               return set(release_reviewer_type);
action
    out   internal           (a : Activity);   --    (See page 8.)
    out   releasable         (d : Data);       --   (See page 14.)
    out   MISSI_releasable   (d : Data);       --   (See page 18.)
    ...
end MISSI_Enclave;
```

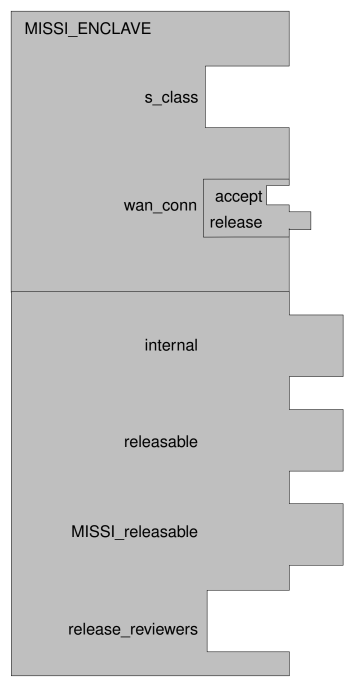


Figure 3: Extending the definition of an enclave

Having identified the *types* of components that make up the architecture, we define their number (if known), their structure (if any) and whether new components can be created while

[1] Some of the other actions and functions will be used later. For each, the comment succeeding the declaration identifies the page on which it is used.

the system evolves, and whether existing components can terminate and remove themselves before the architecture terminates.

In the case of the MISSI reference architecture there is not much structure at the global level, and the architecture does not address the issue of dynamic component creation or removal. In its purest form, we may simply state that the components of the architecture are a *set* of enclaves, a single WAN (a simple routing model) and directory service agent and a set of unclassified (*i.e.*, non-DoD) sites, as in Figure 4.

```
architecture MISSI( ) is
    internet  : WAN;
    DNS       : DirectoryServiceAgent;
    enclaves  : set(MISSI_Enclave);
    sites     : set(Site);
    ...
end MISSI
```

Figure 4: The components of the MISSI reference architecture

This is deceptively simple, but then the architecture *is* rather simple, *at this level*. The complexity arises primarily at the lower level architecture, where we see a wide variety of architecture components and policies.

2.3 How are components connected? Adding structural constraints

Having identified the types and numbers of the components of the architecture, we proceed to define how they may interact. At this level of abstraction, the interaction is quite simple: The enclaves and sites are all connected to the WAN through their respective wan_conn services (Figure 5).

The role of the connection definitions are domain specific. In secure systems architectures, the interpretation of the set of connections would be that they identify *all* possible means of interaction among the architecture components. There is an implied frame axiom for the architecture specification that information shall flow only along those lines and in those forms explicitly defined by the connection definitions for the architecture.

We notice that since all the enclaves are given a bi-directional connection to the internet, we have that the enclaves are all indirectly connected to each other. This is a common pattern – that components of an architecture communicate via intermediaries that allow for communication transformation, filtering, routing, etc. Such intermediaries are called *connectors*.

```
connect
    for e: Enclave in enclaves.enum() generate
        internet.socket to e.wan_conn;
end;
```

Figure 5: Connecting architecture components

2.4 How are connections used? Adding operational constraints

After we have specified the structural properties of the global architecture, we go on to specify some *operational* requirements that implementations have to obey. Operational requirements define protocols and possibly other restrictions on the behavior of components of the architecture. Where a connection between two components indicates a *potential* for interaction, the operational specifications will indicate precisely under what circumstances such interaction actually can (or *must*) take place, as well as indicating when interaction shall *not* occur.

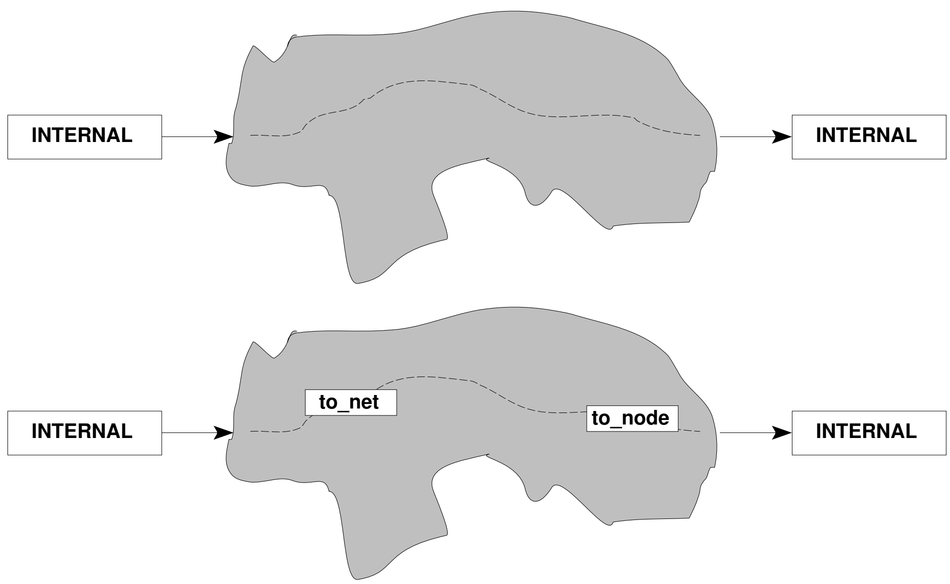
In the constraint sublanguage of the *Rapide* ADL one can specify simple protocols for interaction (such as handshaking, etc.), as well as more sophisticated requirements regarding information flow, causal relationships, etc. At the global level the most powerful security constraint would be that

> *No information should flow from one enclave to another without going through official network connections.*

There are a number of different ways to make such a statement precise, and the *Rapide* formalization of the architecture specification allows us to clearly identify and thus discuss the alternatives. The strictest interpretation is probably that

> *There shall be no internal activity in two distinct enclaves such that they are causally related without intervening* wan_conn *events.*

Stated in *Rapide* (see Figure 6), the semantics may be more immediately apparent: whenever we see a causal chain of events from an internal activity of one enclave to an internal activity of another enclave, then there must be two wan_conn events within that chain, one sending (from the originating enclave), and one receiving (at the other end). The variables ?e1, ?e2 are *free*, indicating that the constraint holds for *all* enclaves.




```
observe (?e1, ?e2: MISSI_Enclave)
    ?e1.internal → any()→ ?e2.internal where ?e1 ≠ ?e2
match
    ?e1.internal → any() → ?e1.wan_conn.to_net
        → any() →
    ?e2.wan_conn.to_node → ?e2.internal;
end;
```

Figure 6: A security constraint

This is a significantly stronger (and to-the-point) constraint than what we would obtain by stating the requirement in terms of *time*. If we interpreted "a → b" as "a happened before b in time" then the above constraint would be satisfied if two enclaves were (legitimately) interacting with high frequency while information were to flow covertly from the one to the other at a lower frequency. The fact that there would be legitimate wan_conn events interspersed between the sending and the receipt of covert information would legitimize the communication of the covert information. On the other hand, the interpretation of "→" as representing *causal* dependency correctly precludes such a scenario from being acceptable.

The *Rapide* pattern language has much in common with regular expressions extended with variables and the ability to evaluate Boolean expressions, and extended to deal with *partial orders* as well as the sequences of more traditional regular expressions. The key difference is that the *Rapide* pattern language encourages specifications of *causal dependency* relationships. The *Rapide* "a→b" relationship between two events requires that they occur in a particular order; a before b, and also that there be an established dependency between a and b, *e.g.* that a represents writing of data and b represents reading of that data, or a represents the sending of a message and b its receipt. For a full exposition of the *Rapide* pattern and constraint languages, see [11, 18, 19, 20]

2.5 Repeat as needed … the *concept of operations* level

The next level of architecture is a *concept of operations* ("*conops*") architecture. The conops architecture specifies the structure of *enclaves*, and how the operations within an enclave are carried out by its various components (including human beings).

As with the global architecture, the definition of the conops architecture identifies (1) the components of an enclave, (2) their connections and (3) how these connections may (or may *not*) be used.

2.5.1 What are the components?

The components are such entities as *users* and *workstations*, *confidentiality* and *authentication servers* as well as other servers such as *firewalls*. We shall not enumerate all the component types of the conops architecture. However, the MISSI document [7] does give us an example of a non-trivial decision we face when formalizing the definitions of the component types. It says:

2(a) *"An authorized releaser for a particular enclave must be a MISSI certificate holder and reside within the enclave."*

This paragraph introduces the component type "*authorized releaser,*" and can be interpreted in two different ways, depending on our interpretation of the word "must." If an authorized releaser *by definition* is a MISSI certificate holder, then one makes the type releaser a subtype of the type certificate_holder. A consequence of such a choice would be that one cannot entertain (or formally specify) situations where a releaser is *not* a certificate holder, just as one cannot entertain the notion that an even number not be an integer.

Another tack would be to identify the relationship between an enclave and its set of releasers, each of which is of the generic MISSI_user_type. In which case we are obliged to define a function from such user components to their set of certificates (in order to state that all releasers hold certificates) as well as a residency relation between enclaves and its residents (in order to state that the residency requirements should hold). Such functions and relations can be defined as being *part* of a component (*i.e.,* an attribute of it), or as a function or predicate external to the component. We chose the latter approach.

We are faced with a similar decision in paragraph 1(b):

1(b) *"All legitimate MISSI users must have a valid certificate for some classification level they are cleared to read."*

Is this a definition of what a "legitimate MISSI user" *is* (in which case we define the type legitimate_MISSI_user and add the requirement that the attribute certificate_set be non-empty)? Or is it a definition of when a MISSI-user is "*legitimate*" (in which case we define the type MISSI_user with the attribute legitimate, which is true if and only if the attribute certificate_set is non-empty)? We settled for the latter interpretation.

```
type Certificate_authority_type is
    include Authority
interface
provides
    function authorized () return Boolean;
requires
    certificate_generator   : Certificate_generator_type;
    ...
end;
```

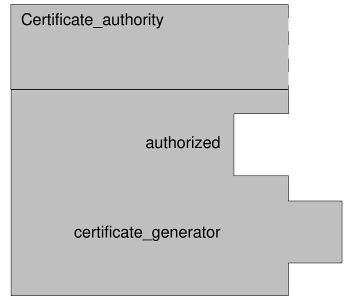


Figure 7: An enclave component identifying its requirements

2.5.2 How are components connected?

At the enclave level we also see a number of requirements regarding access and connectivity, such as:

1(a) *"Authorized certificate authorities (and no others) must be provided with access to certificate generation functions."*

As with many of the MISSI requirements this one has both a prescriptive as well as a restrictive aspect: There shall be access for one class of components, and such access by any other component is prohibited. The former is reasonably interpreted as a structural requirement, the latter may either be structural (that there simply be no physical accessibility), or one of protocol (that there shall be no attempts at exercising the certificate generation functions without proper authorization.)

The prescriptive part of the requirement is easily modeled with in *Rapide* using interface type definitions (see Figure 7). The presence of a **requires** clause in the definition lists all the entities a Certificate_authority_type module expects to be able to use without further ado – it is up to the architecture implementation to supply it with a suitable module to satisfy this requirement. The **requires** section of a type specification indicates what the environment – the *architecture* – has to make available to objects of type Certificate_generator_type. This mechanism differs from the usual object-oriented approach of employing parameterization of the type or the object constructors of the type. If one were to employ the alternative of supplying the server references as parameters to object constructors as in

```
module certificate_authority(certificate_generator : Certificate_generator_type; ... )
      return Certificate_authority_type is
      ...
end;
```

then we would bury a key implicit element of the prose requirements; that the assignment of a server to a user is an *architectural* one, which may change over time as the system evolves and the user acquires or relinquishes certificates.

Rapide allows us to make the style distinction between parameterized *definitional dependencies* (which are identified by the parameter lists of type definitions), parameterized *implementation dependencies* (which are identified by the parameter lists of module constructors) and (*dynamic*) *architectural dependencies* (which are identified by **requires** sections in interface definitions).

The restrictive part of the requirement ("*…and no others…*") can be addressed explicitly or implicitly. By using the frame axiom for security architecture conformance (*i.e.*, in the absence of any connections, no information flow shall take place) we can deduce this restriction from the absence of any explicit connections between modules that are *not* authorized certificate authorities and certificate generators. Such a *structure-oriented* representation of the requirement would be using conditional connections in the architecture itself to set up the connections for all the authorized certificate authorities (see Figure 8). Here the *architecture specification* makes clear that access to the **new_token** function will be given only to those **certificate_authority_type** components that have the **authorized** attribute set to true.

```
connect
    (?c : Certificate_authority_type)
        ?c.new_token where ?c.authorized
    to  certificate_generator.new_token;
```

Figure 8: A conditional connection

However, a requirements document that relies on the *absence* of certain statements might be asking for too much of the reader.

If one instead wishes to make this requirement explicit in the formal version of the reference architecture then it is naturally rephrased as a *protocol requirement*; that all modules attempting to make use of the certificate generators are duly authorized. Since this is a usage restriction relevant to certificate generators, it is reasonable to locate it within the definition of the **Certificate_generator** interface (see Figure 9).

When it states *"Authorized certificate authorities (and no others)…"* the constraint interprets the "*(and no others)*" as meaning not only all non-authorized certificate authorities, but also all other entities of other categories. The mechanism is through observing *all* calls to the **new_token** function, and then requiring that all these calls be made by components of the **Certificate_authority_type**, where that component also has the **authorized** attribute set to true.

```
type Certificate is interface ... end;

type Certificate_generator_type is interface
        function new_token(...) return Certificate;
    …
constraint
    observe (?p : root) new_token'call(performer is ?p)
    match (?c : Certificate_authority_type)
        new_token'call(performer is ?c) where ?c.authorized;
    end;
    …
end certificate_generator_type;
```

Figure 9: A restrictive protocol definition

A number of the requirements – 1(c,d,e) – as well as the later 1(e, g, h, i, k), are on the same form:

> *"All MISSI certificate holders must be provided with access to appropriate* <keyword> *functions for each classification level they are cleared to read."*

(Where the *<keyword>* identifies the distinct functions, such as *confidentiality*, *integrity*, and *certificate validation*.)

There are two elements to each of these requirements as well:

1. There is a reference to what a confidentiality (and similarly integrity-, certificate validation-, etc.) function *is*. That aspect deals with definitions of functions and abstract data types, and are best dealt with using an ADT- or object specification formalism. *Rapide* incorporates the data type specification capabilities of ANNA [], but since the specification of datatypes impinges minimally on our discussion of architectures, we shall not pursue this aspect beyond the sketch of a *Rapide* definition of a **Wrapper_type**, with a specialization to a **Confidentiality_server_type** (Figure 10).

2. That for a particular functionality the actual function supplied may differ depending upon which access level is being exercised by the certificate holder. Consequently, access to server functions may change over time, as certificates are acquired or relinquished. Furthermore, there is no requirement that the appropriate function for a given access level be fixed for the duration of the system – consequently, the formalization should allow for a conforming system to supply different functions at different times for a given access level and user.

To state or allow for the latter is a challenge to ADLs and specification formalisms based on (first order) logics, which do not address the issue of *time*. In *Rapide* time is implicitly present throughout a specification, and can be made explicit as necessary through references to clocks or events.

We shall assume (see 1 above) that we can define precisely what is expected of a set of *confidentiality functions* (and similarly for the other functionalities).

```
type Wrapper_type (type content_type, packaged_type, key_type)
    constraint content_type <: equal
is interface
provides
    function wrap      (k : key_type; c : content_type)      return packaged_type;
    function unwrap    (k : key_type; p : packaged_type)     return content_type;
    function check     (r: root)                              return Boolean;
constraint
    forall c : content_type; p : packaged_type; k : key_type =>
        check(wrap(k, c)) and
        not check(unwrap(k, p)) and
        exist kk : key_type => unwrap(kk, wrap(k, c)) = c;
end;

type Confidentiality_server_type (type key_type, wrap_info_type, wrapped_type)
    constraint ...
    – – constraints on the parameter types as to minimum information required[2]
is
    include Wrapper_type (key_type, wrap_info_type, wrapped_type)
interface
    – – constraints specific to the confidentiality server
    – – in addition to the included wrap, unwrap and check
end;
```

Figure 10: Wrappers and unwrappers, and a specialization to a confidentiality server type

Given the definitions of the server functions, we specify the access requirements explicitly (Figure 11). Each MISSI_user_type object will assume the (external) existence of a function returning a reference to a confidentiality server (assuming that the types key_type, wrap_info_type, and Wrapped_type are defined elsewhere), an integrity server and a validation server.

This requirement is formalized using the **requires** clause of *Rapide*. In so doing we signal that a MISSI_user_type object may call the function confidentiality_server with the expectation that the architecture (*i.e.*, the environment) will supply a binding for it. The architecture may change this binding during the execution of the system. By adding the "**constraint** (classification.element(c))" to the function declaration we identify that the function is only required and accessible for a particular classification level if the MISSI_user actually is cleared at that level.

The "*(and no others)*" part of requirements 1(j, k) are dynamic prohibitions and are formalized in the same way we made precise the similar injunction in 1(a) (see page 11), *i.e.*, as a check

[2] These constraints would indicate whether the user identity would have to be included in the key or wrap_info types, etc.

that whenever there is a call for a confidentiality_server it is from a component with the proper clearance.

```
type confidentiality_ref is Confidentiality_server(Key_type, Wrap_info_type, Wrapped_type);
    -- and similarly for the other servers

type MISSI_user_type is interface
provides
    classification : set(Classification_type);
...
 requires
    function confidentiality_server  (c: Classification_type) return Confidentialy_ref
              constraint (classification.element(c));
    function integrity_server         (c: Classification_type) return Integrity_ref
              constraint (classification.element(c));
    function validation_server        (c: Classification_type) return Validation_ref
              constraint (classification.element(c));
    ...
end MISSI_user_type;
```

Figure 11: Capturing access requirements

2.5.3 How are connections used?

Finally, there are the policy requirements, stating preconditions for information flow within the enclave or from the enclave to the outside. An example is

2(c) *"All data transferred outside of a secret-high enclave and addressed to a MISSI certificate holder must be protected by a confidentiality service, a proof of origin non-repudiation service and a recipient authentication service."*

This can be modeled either as the data having certain properties (essentially having stamps of approval from the respective servers), or as a precondition on the *history* leading up to a release of data outside a secret-high enclave. We recommend the latter approach (see page 19), in which case we make use of the *Rapide* pattern language to identify the protocol that defines a data release: it fits the pattern of Figure 12, *i.e.*, that for any piece of data, if it is released to the outside then that release has to be preceded by the three services checking it off.

```
pattern outside_release_ok(?d : data) is
    (conf_service(?d) ~ origin_service(?d) ~ recip_service(?d)) → data_release(?d)
end;
```

Figure 12: Abstracting patterns

2.6 Defining relationships between architectures

At this point in our process we have a definition of the global level of the reference architecture, whose principal components are MISSI_enclaves and WANs, and the conops level, whose principal components are workstations, firewalls, LANs, and servers.

Part of the definition of a reference architecture with multiple levels of abstraction identifies precisely how the levels are related. There are clear relationships between these two levels – *e.g.*, the enclave architectures of the lower level are modeling the MISSI_enclaves at the top level, the activities of the firewalls at one level represent release and accept events at the higher level, the simple wan_conn of the abstract enclave definition corresponds to the firewall_type objects of the conops architecture. But in the conops level definition there is no action "internal" which may play such a crucial role in the constraints of the global level architecture – the reference architecture must define what conops-level events correspond to the internal events of the global level.

```
map abstract_enclaves from e: enclave_architecture to MISSI_enclave is
rule
rule_1:
        (?e : event) ?e@
    ||> internal(?e);

rule_2:
        (?d : Data, ?a : Address) @firewall.wan_conn.to_net(?d,?a)
    ||> wan_conn.release(?d,?a);

rule_3:
            (?ws : COTSWorkstation; ?content : Data)
            ?ws.net_conn.to_node(Certificate_Validation, ?content)
        ~
            ?ws.net_conn.to_node(Integrity, ?content)
        ~
            ?ws.net_conn.to_node(Encryption,?content)
    ||>
        releasable(?content);;
    ...
end;
```

Figure 13: Three abstraction maps

It would not be a good idea to merge the definitions from the two levels into one unstructured definition of the notion of "enclave." Instead we use *Rapide maps* to relate components and activities of the conops architecture to their corresponding components and activities in the global architecture.

Figure 13 gives an example of such an abstraction map. It consists of three *rules*, each of which defines how occurrences of patterns of events at the conops level correspond to more abstract events at the global level.

The first rule indicates that *any* event in the conops enclave ("(?e : event) ?e@") will be mapped up to ("||>") the abstract internal event, indicating that something happened (but where we abstract away from the particulars of what happened). The second rule maps each transmission of data from the firewall to the WAN ("@firewall.wan_conn.to_net") to the abstract event release, representing the flow of information out of the enclave, abstracting away the particulars of how the information became public. The last rule is an example of how a more complex pattern of events may represent a single abstract event: Whenever a piece of information (represented by the placeholder ?content) has been approved by the validation, integrity, and encryption servers then the information becomes releasable, abstracting away from the actual protocol required for attaining this status.

Figure 14 shows an excerpt from a computation, indicating the two levels of abstraction and the relationship between a set of events at the lower level with a single abstract event at the higher.

As we see, there is no prohibition against a single concrete event participating in more than one abstract event (as each of the server events are both represented as abstract internal events as well as being part of the releasable event).

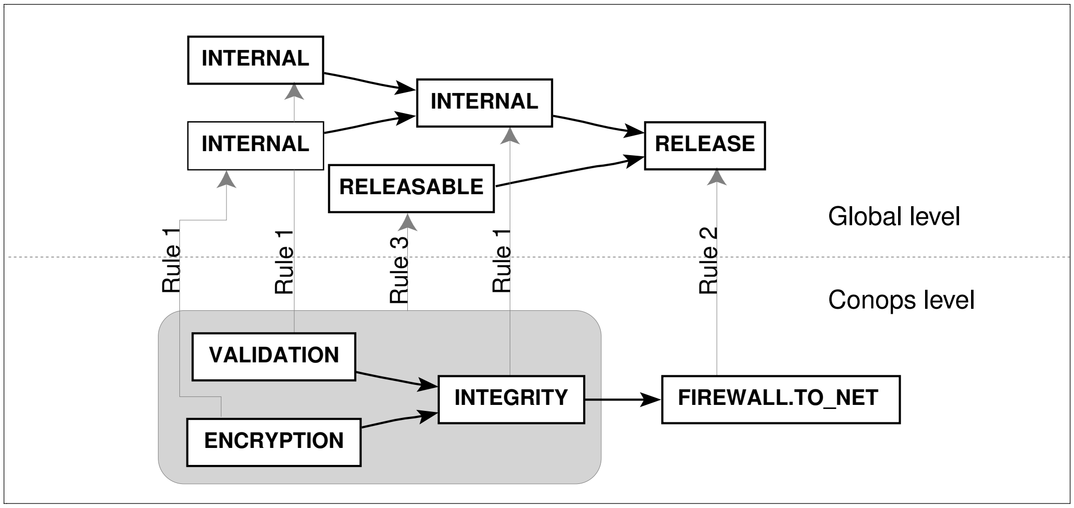


Figure 14: Events at two levels of architectural abstraction

If one of the steps in the protocol is missing (for instance, if the `Validation` never took place), we would not get the required `Releasable` global event. The result would be as in Figure 15, and would result in a violation of a global level constraint (see page 17).

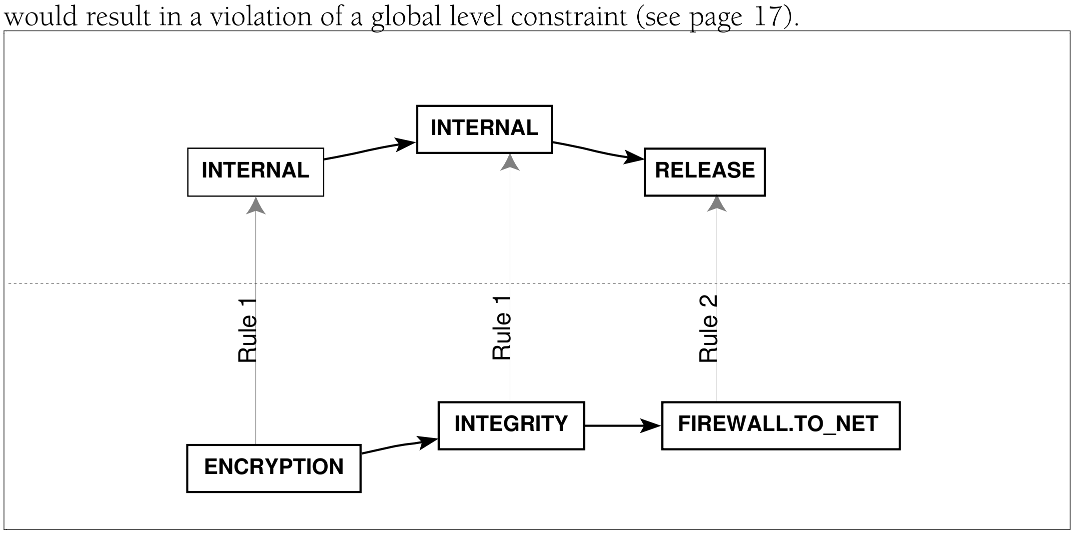


Figure 15: Missing conops event → missing global event

3. Formalizing the MISSI requirements summary

In this section we go through all the requirements of the MISSI overview, showing how we would capture them in *Rapide*.

We have already dealt with the very first requirement (section 2.5.2, page 13):

1(a) *"Authorized certificate authorities (and no others) must be provided with access to certificate generation functions."*

We have also touched upon the next requirement earlier (section 2.2):

1(b) *"All legitimate MISSI users must have a valid certificate for some classification level they are cleared to read. Entities with valid certificates must be legitimate MISSI users."*

If this is a definition of when a MISSI-user is "*legitimate*" we define the type MISSI_user with the attribute legitimate, which is true if and only if the attribute "certificate_set" is non-empty (see Figure 16.[3]

The last constraint implies the first, of course, but in the interest of clarity of intention we state both explicitly, since redundancy adds rather than detracts from the confidence we have in the specification.

An alternative representation would define two types; MISSI_user_type and legit_MISSI_user_type <: MISSI_user_type. The latter would be constrained always to have in hand appropriate certificates, the former would allow its transformation into a legit_MISSI_user_type object after performing the appropriate checks.

The next three requirements – 1(c,d,e) – as well as the later 1(e, g, h, i, k), all contain a requirement on the same form:

> *"All MISSI certificate holders must be provided with access to appropriate* <keyword> *functions for each classification level they are cleared to read."*

(Where the *<keyword>* identifies the distinct functions, such as *confidentiality*, *integrity*, and *certificate validation*.) They have been discussed extensively earlier, in section 2.5.3.

```
type MISSI_user_type is interface
provides
  function classification  ()  return set(Classification_type);
  function certificates ()  return set(Certificate_type);
  function legitimate      ()  return Boolean;
  …
  function residency       ()  return Enclave;
  ...
constraint
  legitimate() = not certificates().empty;
  legitimate() implies
  not map(certificate_type,classification_type,certificates(),security_level).intersect(classification()).empty;
end MISSI_user_type;
```

Figure 16: An invariant constraint

Requirements 1(j, k) strengthens the access requirements by adding that accessed functionality be

> *"…for the enclave in which they reside. (All* <entities> *are MISSI certificate holders and reside in the enclaves in which they perform their task.)"*

These are simply invariants over the relationships between components and enclaves, and could be stated in those terms, *e.g.*, in the subtype release_reviewer_type of the MISSI_user_type there is the invariant that:

```
…
not certificates().empty;
residency().release_reviewers().element(self);
…
```

---

[3] The polymorphic function map takes two types S and T (the source and target type), an object M of type set(S) and a function F with signature S→T, and returns an object of type set(T), each of whose elements is the result of applying F to some element of M. The function security_level is assumed to map certificates to security levels.

[4] Each shaded area represents a releasable event justifying the corresponding release event. There is an example of a single releasable justifying multiple releases, as well as a single release being justified by multiple releasable events.

Sections 2 and 3 of the requirement set identify the circumstances under which information may be released from or accepted into an enclave.

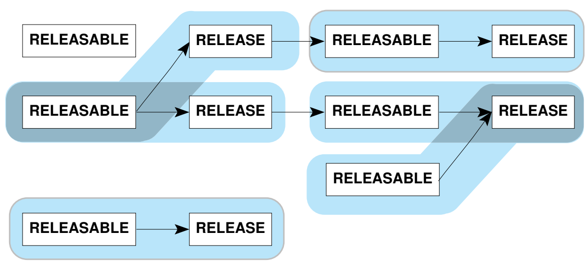
2(a) *"An authorized releaser for a particular enclave must be a MISSI certificate holder and reside within the enclave."*

2(a) is similar to the requirements of 1, and is dealt with in the same way.

2(b) *"All data transferred outside of a secret-high enclave must have been sent by an authorized releaser in the originating enclave, must be protected by an integrity server, and must pass a releasability check in the originating enclave."*

2(b) establishes protocol precursors for the event representing the release of data from an enclave. Assuming that data is being released by means of the firewall communicating to the network, the notion of data being *releasable* was captured on page 14. Given that, 2(b) becomes a constraint of the abstract **enclave** definition. Observing **release** and **releasability** events (Figure 17), every communication to the net of a piece of data has to be preceded by a **releasability** event (but not the other way around – releasable data is not required to actually be released):

Note that there must be a *causal* chain from establishing releasability to the actual release.




```
observe
 from releasable, release
 where (security_classification() = secret_high)
match (?content : Data)
        [* rel union] (releasable(?content) →
            release(?content)) union
            [* rel ~] releasable;
end;
```

Figure 17: Satisfying the releasability requirement[4]

The use of the **union** relation over the set of pairs of **releasable** and **release** events allows a single **releasable** event to justify multiple actual releases (as in Figure 17).

If the requirement specified that all releasable data actually be released then we would omit the second component of the union collecting all the dangling **releasable** events.

2(c) *"All data transferred outside of a secret-high enclave and addressed to a MISSI certificate holder must be protected by a confidentiality service, a proof of origin non-repudiation service, and a recipient authentication service."*

2(c) is similarly structured to 2(b), the main difference being that we limit our interest to data addressed to MISSI certificate holders. By implication, this requires a global (specification) function mapping addresses to attributes of the addressee[5]. Figure 18 gives a variant on the 2(b) requirement. The global event **MISSI_releasable** is defined in Figure 19, and is similar to the definition of **releasable** (see Figure 13 on page 14), as a mapping from a protocol pattern at the conops level to a single event at the global level. We assume that the function **Recipient** : **Data→Root** gives us the identity of the intended recipient of the data, and then use subtyping to limit the applicability of the mapping to those messages that have **MISSI_user**s as recipients.

[5] This mapping seems methodologically dubious, but it does not offer any problems for the transformation of the prose into precisely formalized requirements.

```
observe
    from MISSI_releasable, wan_conn.release where (security_classification() = secret_high)
    match (?content : Data)
                [* rel union] (MISSI_releasable(?content) → wan_conn.release(?content))
        union   [* rel ~] MISSI_releasable;
end;
```

Figure 18: MISSI releasability restriction

```
rule
    (?ws : COTSWorkstation; ?content : Data)
            (?ws.net_conn.to_node(Confidentiality, ?content)
        ~
            ?ws.net_conn.to_node(Non_repudiation, ?content)
        ~
            ?ws.net_conn.to_node(Recipient_validation, ?content)
||>
        MISSI_releasable(?content);;
```

Figure 19: A variant on the releasability definition

2(d) *"If a recipient is capable of providing authentic receipts and the originator of the data requests a receipt, all data transferred outside of a secret-high enclave must be protected by a proof of receipt non-repudiation service."*

This requirement mixes references to capabilities of enclaves (offering an authentication service) and events (the data being transferred with a return receipt request). To be "receipt confirmation capable" is modeled by adding a node Receipt_authentication_enclave to the type structure, introducing a subtype of the Enclave type. Stated in protocol terms, a receipt acknowledgment must be generated whenever data leaves a secret-high enclave addressed to a receipt confirmation capable component. There are a number of ways one can phrase this. As a negative, one can write that for each release event and all its (causally) subsequent acknowledgments for the receipt of the release, the set of acknowledgments cannot be empty (Figure 20).

```
observe (?content : Data; ?recipient : receipt_authentication_enclave; ?address : Address)
    wan_conn.release(?content, ?address)
        where (security_classification() = secret_high and ?recipient = ?address.enclave),
    → ([* rel ~] receipt_acknowledge(?content.ack))
not match
    wan_conn.release;
end;
```

Figure 20: A negative form of constraint 2*(d)*

Or one can write it in positive terms – for each release event and all its (causally) subsequent acknowledgments for the receipt of the release, the set of acknowledgments has to contain at least *one* acknowledgment (Figure 21).

```
observe (?content : Data; ?recipient : Receipt_authentication_enclave; ?address : Address)
    wan_conn.release(?content, ?address)
        where (security_classification() = secret_high and ?recipient = ?address.enclave),
    → ([* rel ~] receipt_acknowledge(?content.ack))
match
    wan_conn.release → ([+ rel ~] receipt_acknowledge);
end;
```

Figure 21: A positive form of constraint 2*(d)*

In both cases, the ***Rapide*** form is one of (1) filtering the set of events to extract those subsets (possibly overlapping) that are of interest (in this case to each single release and its (possibly empty) set of responding acknowledgments), and then (2) specifying the pattern these events have to comply with (in this case that the set of acknowledgments be non-empty).

3(a) *"An authorized receiver for an enclave must be a MISSI certificate holders and reside within the enclave in question."*

3(a) is similar to 2(a), and is dealt with in the same way.

3(b) *"Any data admitted to a secret-high enclave from the outside must be protected by an integrity service, must pass an admissibility check for the enclave, and must have a designated recipient within the enclave who is authorized to receive external data."*

3(b) is similar to 2(b), and is dealt with in the same way.

4(a) *"All sensitive administrative data must be protected by an integrity service while in transit or in storage."*

As with 2(b) and (c) there are two, quite distinct, perspective on this kind of constraint.

One can either view the requirements as related to *state*, *i.e.,* every piece of (administrative) data has some state attribute indicating whether it is in storage, in transit or in (possibly) other modes. In which case the natural mode of expression is one of first order logic (as in [7]), but at the cost of reduced checkability and increased complexity of expression – data and other basic types would acquire an ever-growing set of more or less obvious attributes, an attribute collection which may become intractable as the abstract notion of data becomes refined.

Or one can view it more dynamically, and focus on the *action* of storing or putting into transit a piece of data, in which case the assertion of being protected by an integrity service is tied to the transitional event itself. This is the path taken in the formalization of 2(b) and (c), and would be repeated for 4(a), here.

4. Putting a *Rapide* reference architecture to use

Given a ***Rapide*** formalization of the reference model we can put it to a number of different uses. The most obvious is as a precise definition of the model itself – being expressed in a formal language it allows us to draw unambiguous conclusions from the formalization based on testable arguments within a formal framework (in the case of ***Rapide*** constraints the framework is a simple one of sets and partial orders).

Since ***Rapide*** is supported by a growing toolkit of visualization and testing modules [21, 22], the reference architecture can be the target for *conformance testing* by implementations purporting to satisfy the architecture's requirements. Such automatic conformance testing requires two things:

- An instrumentation of the implemented system which supplies the tools with the information required to compare the implementation to the reference architecture. Such an instrumentation can in many cases be automatically generated by a modified set of compilers,[6] generating the code necessary to create events and maintain the dependency graph.
- An abstraction *map* essentially defining how the patterns of events generated by the instrumentation correspond to the types of events and components referred to in the architecture.[7]

  Such a map makes the conformance argument precise, and adds documentation as to how the implementor thought her system relates to the reference architecture.

---

[6] Such an instrumented compiler-set exists for Java, Verilog and CORBA IDL besides for ***Rapide*** itself.

[7] We have already made use of such maps in defining how the abstract releasable event occurs as an abstraction from a pattern of lower-level events.

Given such instrumentation and the argument how conformance is obtained, the system conformance test becomes automatic, and can become a standard part of any regression test one might wish to subject the system to as its implementation evolves.

Furthermore, the instrumentation together with its conformance map can become an embedded, permanent part of the production system. The result is another layer of security checking, where the different perspective on the system offered by the conformance argument may detect architecture violations that might otherwise go unnoticed.

A variant of the conformance testing is the use of the tools for *scenario* testing and presentations. The *Rapide* toolkit has been applied to such diverse models as the SPARC V9 reference hardware architecture and a stock market model, as well as a simple scenario for security protocols based on elements of the MISSI reference architecture.

In the security model scenario we constructed a model vertically partitioned into three layers.

At the bottom layer we defined an executable conops model of *users*, *workstations*, *protocol servers*, *firewalls*, and *networks*.

The topology was one of a set of LANs, each with its workstations, firewalls and servers, and each workstation with its users. The LANs were connected by means of a WAN, through their respective firewall modules.

All the networks were broadcast networks.

This bottom layer corresponds to an actual system, a flat, relatively unorganized set of components communication hither and thither – possibly in conformance with the requirements of the reference architecture. Or possibly not – that is what the toolkit checks.

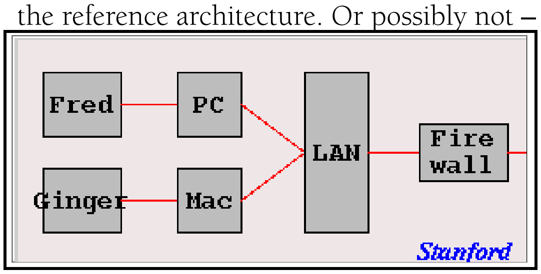


Figure 22: Some components of an intermediate level model architecture

The second level is an intermediate one. Each architecture is an *enclave*, each of which is accompanied by a set of the enclave-related requirements (such as 2(b), about releasability). Each enclave in the intermediate architecture is the target for a *Rapide* map, which transforms patterns of conops model behaviors into activities defined for enclaves (*e.g.,* as in the definition of the releasability map, see Figure 13). Some components of an enclave is shown in Figure 22 (from an animation of the conformance check), an enclave with two users, two workstations, a LAN and a firewall (besides the local servers, not shown in this figure).

The third level is that of the *global* architecture, consisting of *enclaves, WANs,* etc. (Figure 23 gives an abbreviated view, from an animation of the reference architecture conformance test of a model with four enclaves.) At this level we check the constraints relating to multi-enclave concerns, such as the global requirement of page 8, prohibiting covert channels. The architecture level can be obtained by maps directly from the conops model, or in two stages: by the maps from the conops model to the intermediate level, and then maps from the intermediate level on to the global level. Which of these one chooses is a question of whether the intermediate models contain all the information required for the global architecture model (*e.g.,* the notion of general internal activity) or not.

A model (or a system in testing or production) typically generates a large number of events. When investigating data for possible non-conformance it is critical that the number of data elements – events of possible interest – be reduced as early as possible. The *Rapide* toolkit offers two means to achieve this end. The first is the use of architecture maps in structuring the instrumentation. Each map construct results in the automatic construction of a *transformational filter* (or *sieve*), which passes on only those events that are considered significant in the abstraction, possibly transformed so as to aggregate event patterns into single events or simpler event patterns.

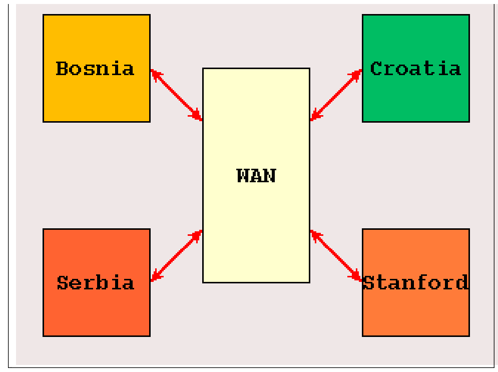


Figure 23: A global level architecture

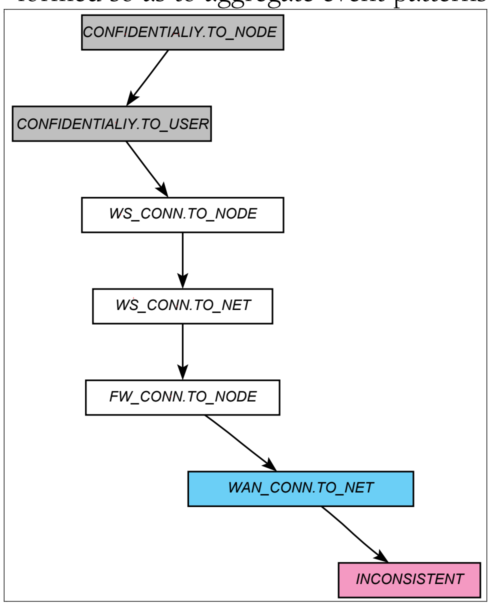


Figure 24: Detecting a protocol violation

The second is the visualization toolset of *Rapide*. This part of the toolset allows the user to apply various patterns of events to a given execution, displaying only those events fitting patterns of interest. Combined with the *Raptor* [22] animator this makes it possible to watch an animation of a running system at a chosen level of abstraction. Then, if interesting events (such as protocol violations) are detected, the user can move to the POV (poset visualizer) [21] and use it to investigate the causal patterns leading up to the events that piqued her interest. In particular, the POV allows the efficient removal of extraneous information, to ease the identification of interesting events among the clutter of all the events of the system.

As an example, consider the events of Figure 24. These were culled from the execution of a network model, after the occurrence of an *inconsistent* event was observed at the global level. (An *inconsistent* event signals the system's detection of a constraint violation, in this case the global releasability constraint of Figure 18, page 18). By moving from the global architecture to the conops architecture, using the POV, and then following the causal links past-wards from the *inconsistent* event, we identify its cause: the absence of the Integrity and Encryption steps of the protocol making a piece of information releasable. As the user only engaged the Confidentiality server, once the information was transmitted from the firewall to the WAN, she was in violation of the reference architecture constraints.

5. Conclusion

We have indicated how one may use the event based language of *Rapide* to capture elements of a reference architecture. Both the structural and the operational requirements of the architecture can be stated precisely in *Rapide*, and the resulting specification may become the basis for (1) analysis, (2) model checking, (3) implementation conformance testing and (4) production code conformance surveillance.

A key element in the successful application of an architecture description language to the design of reference or other software architectures is the degree to which it allows one to state *all* aspects of the architecture, and the flexibility of the *abstraction* mechanisms that may be applied when the conformance requirements are stated (as part of the architectural design). Distinct architectural perspectives require distinct abstraction mappings, and it is important that the designer be able to separate such perspectives from each other – giving separate reference architectures for each perspective, as appropriate.

Furthermore, an ADL is only as good as the tools that support it – in the absence of tool support, design capture and conformance reasoning easily devolves into vague hand-waving. The tool support should help automate conformance testing and other aspects of architecture design analysis, as well as allowing the designer to construct test scenarios and visualize the behavior of architecture conforming systems.

We have found that the *Rapide* ADL with its supporting toolset offers an interesting approach to the design of distributed architectures. In particular, the event orientation of the system, coupled with its sophisticated ability to identify causal chains and patterns of behaviors where causal relationships may play an integral role are quite enticing.

6. References

1. Allen, R., Garlan, D.: Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*. IEEE Computer Society Press, May 1994.
2. Boehm, B. W.: Software Process Architectures. In *Proceedings of the First International Workshop on Architectures for Software Systems.* Seattle, WA, 1995. Published as CMU–CS–TR–95–151.
3. Garlan, D.: Research directions in software architectures. *ACM Computing Surveys,* 27(2): 257–261. 1995.
4. Garlan, D., Shaw, M.: *An Introduction to Software Architecture.* Volume I. World Scientific Publishing Company, 1993.
5. Ellis, W.J. et al.: Toward a Recommended Practive for Architectural Description. In *Proceedings 2nd IEEE International Conference on Engineering of Complex Computer Systems,* Montreal, Canada, 1996.
6. Gennart, B. A., Luckham, D. C.: Validating Discrete Event Simulations Using Pattern Mappings. In *Proceedings of the 29th Design Automation Conference (DAC)*, IEEE Computer Society Press, June 1992, pp. 414–419.
7. Johnson, D. R., Saydjari, F. F., Van Tassel, J. P.: MISSI security Policy: A Formal Approach. R2SPO Technical Report R2SPO–TR001–95, NSA/Central Security Service, July 1995.
8. Luckham, D. C.: *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*, Springer-Verlag, Texts and Monographs in Computer Science, October, 1990.
9. Luckham, D. C., Vera, J., Meldal, S.: *Key Concepts in Architecture Definition Languages.* Submitted to the CACM. Also published as technical report CSL-TR-95-674, Stanford University, 1996.
10. Luckham, D.C., Vera, J.: An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(3):253–265, June 1993.
11. Luckham, D.C.: Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events, DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996.
12. Meldal, S.: Supporting architecture mappings in concurrent systems design. In *Proceedings of the Australian Software Engineering Conference*. IREE Australia, May 1990.

13. Meszaros, G.: Software Architecture in BNR. In *Proceedings of the First International Workshop on Architectures for Software Systems*. Seattle, WA. 1995. Published as CMU–CS–TR–95–151.

14. Moriconi, M., Qian, X.: Correctness and composition of software architectures. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*. New Orleans, LA. December 1994.

15. Mitchell, J.C., Meldal, S., Madhav, N.: An Extension of Standard ML Modules with Subtyping and Inheritance. In *Proceedings of the 18th ACM Symp. on the Principles of Programming Languages,* ACM, ACM Press. 1991, pp. 270-278. Also published as Technical Report CSL-TR-91-472, Computer Systems Laboratory, Stanford University.

16. PAVG: The *Rapide* Architecture Description Language Reference Manual. <http://anna.stanford.edu/rapide/lrms/architectures.ps>

17. PAVG: *Rapide* toolset information. <http://anna.stanford.edu/rapide/tools.html>

18. PAVG: *Rapide* Examples. In preparation.

19. PAVG: The *Rapide* Pattern Language Reference Manual. <http://anna.stanford.edu/rapide/lrms/patterns.ps>

20. PAVG: The *Rapide* Constraint Language Reference Manual. In preparation.

21. PAVG: *POV–a partial order browser*. <http://anna.stanford.edu/rapide/tools-release.html>

22. PAVG: *Raptor–animating architecture models*. <http://anna.stanford.edu/rapide/tools-release.html>

23. Santoro, A., Park, W.: *SPARC-V9 architecture specification with Rapide*. Technical report CSL, Stanford University (to appear).

24. Soni, D., Nord, R.L., Hofmeister, C.: Software Architecture in Industrial Applications. In *Proceedings of the 17th International Conference in Software Engineering*. ACM, April 1995.