Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures

R. A. Riemenschneider
Computer Science Laboratory
SRI International
Menlo Park, CA 94025

ABSTRACT

Architectural definition often begins with a very high-level architectural description, e.g., specification of dataflow among a few abstract components. This high-level description is then refined in order to ultimately achieve a much more detailed description that can be implemented directly. On the other hand, with the advent of the object-oriented paradigm, architectural definition can consist of assembling a collection of standard components and connectors. But, if such an assemblage is complex, it is desirable to generate more abstract descriptions from it, in order to facilitate understanding and analysis. In either case, the end product of the architecting process is an *architectural hierarchy*, a collection of architectural descriptions linked by mappings that interpret the more abstract descriptions in the more concrete descriptions.

Formalized transformational approaches to architecture refinement and abstraction have been proposed. One argument in favor of formalization is that it can result in architectural implementations that are guaranteed to be correct, relative to the abstract descriptions. If an abstract description and an implementation are correct with respect to one another, conclusions obtained by reasoning from an abstract architectural description will apply to the implemented architecture as well. But this correctness guarantee is achieved by requiring that the implementor use *only* verified transformations, i.e., transformations that have been proven to produce correct results whenever they are applied.

This paper explores an approach to allowing the implementor the freedom to use transformations that have not been proven to be generally correct, without voiding the correctness guarantee. *Checking* means determining that application of the transformation produced the desired result by performing a straightforward calculation of some sort. Checking allows the use of transformations that have not been generally verified, even transformations that are known to sometimes produce incorrect results, by showing that they work *in the particular case.*

Our initial approach to checking, inspired by Necula and Lee's work on certifying compilers, involves checking that these non-verified transformation steps preserve proofs of the critical properties of interest, such as system security. This approach will be illustrated by an example drawn from our research on architectures for secure distributed transaction processsing. Other approaches, useful for checking that broad classes of properties are preserved by the transformation step, are also being investigated.

Although the primary objective of this work is to allow the architect greater flexibility in correctly implementing abstract architectures and generating accurate abstract views of architectures, it has other benefits as well. For example, because verification is a more complex and error-prone process than checking, a combination of verification and checking particular refinement steps can substantially increase confidence that an abstract description correctly characterizes an implementation.

1 Introduction

The process of specifying an architecture often begins by providing a very high-level description of it. This description characterizes the architecture in terms of a few abstract components, perhaps the principal functions the system must perform and some data stores. These components are linked by abstract connectors, perhaps indicating dataflow or control flow relationships among the components. This abstract description provides an easily understood overview of the entire system architecture, but omits so much detail that it provides relatively little guidance to someone charged with implementing the architecture using programming language level and operating system level constructs. So the abstract description must be successively refined — with complex components and connectors decomposed into simpler parts, and abstract specifications of operations and relationships replaced by more concrete specifications — until an appropriate amount of detail has been added. It usually is desirable to continue the refinement until implementation-level constructs have replaced all the abstractions.

Alternatively, architecting a system can consist of assembling instances of reusable component and connector types selected from a library. Such libraries effectively make implementation-level more abstract, and reduce the conceptual gap between the requirements specification and the implemented architecture. Nevertheless, combining a large number of components and connectors in complex ways can easily result in an architecture that is hard to understand and analyze. So, it is desirable to generate more easily comprehensible abstract representations of the implementation-level architecture.

In either case, the end product of the architecting process is typically a collection of architectural descriptions, at different levels of abstraction and often in different styles [5]. The more abstract descriptions are linked to the more concrete descriptions by interpretation mappings. An interpretation mapping says how the abstractions are implemented.[1] It sends each sentence in the language of the abstract description to a corresponding sentence in the language of the concrete description. For example, the fact that some component a is implemented by components $\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n$ would be indicated by mapping the sentence

$$\mathsf{Component}(\mathsf{a})$$

to the sentence

$$\mathsf{Component}(\mathsf{a}_1) \wedge \mathsf{Component}(\mathsf{a}_2) \wedge \ldots \wedge \mathsf{Component}(\mathsf{a}_n)$$

The collection of architectural descriptions and interpretation mappings that comprise the complete architectural specification is called an *architecture hierarchy.*

There are many advantages to formalizing refinement and abstraction in system development: a library of refinement or abstraction transformations provides a "corporate knowledge base" of standard, or preferred, development patterns; mechanizing the application of these transformations lessens the likelihood of clerical errors during the development process; reuse of the transformations will result in greater validation of the patterns they codify; and so on. But one of the most fundamental advantages of formalization is that it allows the average developer to produce abstraction hierarchies that are guaranteed to be consistent. In other words, the use of verified transformations in the development process will guarantee that abstractions accurately characterize implementations, albeit more abstractly. A verified refinement transformation is one that has been proven to produce a correct implementation of whatever it is applied to. A verified abstraction transformation is one that has been proven to produce a correct abstraction of whatever it is applied to.

Even if attention is restricted to the case of architectures, there is some debate as to exactly what *correct* should mean. We have proposed a somewhat stricter than usual criterion for correctness [12], while others have argued that the standard criterion is preferable [19]. For present purposes, any reasonable criterion that characterizes correctness in terms of preservation of truth will do perfectly well. The standard correctness criterion is that every consequence of the abstract description must be a consequence of the concrete description as well. More precisely, for every sentence $\mathbf{A}$ in the language of the abstract description,

$$T_1 \vdash \mathbf{A} \quad \Longrightarrow \quad T_2 \vdash \mu(\mathbf{A})$$

where T_1 is the logical theory that formalizes the abstract description, T_2 is the theory that formalizes the concrete description, and μ is the interpretation mapping that links the two theories.[2] A mapping μ that satisfies this condition is called an *interpretation of* T_1 *in* T_2. Our proposed stronger criterion for purely structural descriptions replaces the conditional with a biconditional, i.e., requires that the interpretation mapping be a *faithful theory interpretation.* One might also employ weaker-than-standard criteria, where only some consequences of the theory — properties of special interest — need be preserved.

What all these criteria have in common is that they justify the use of formal reasoning about the architecture based on the more abstract descriptions. If some sentence is shown to be a formal consequence of the abstract architectural theory, the concrete theory is known to correctly implement the abstract theory, and the sentence is among those that the correctness criterion guarantees are preserved by the implementation, then the sentence is known to be a consequence of the concrete theory as well. It is correctness guarantees that link the results of abstract analyses to the real world.

The usual approach to producing a correctness guarantee is restricting the architect to the use of verified transformations. This approach suffers from a problem, in practice. Even given a fairly mature library of verified transformations, it would hardly be surprising if an architect found himself unable to perform a certain refinement or abstraction step that he believed to be correct because the required transformation has not been included in the library. Expecting the typical system architect to produce a formal proof that the step is correct is unrealistic, yet the presence of a single unverified implementation step in the hierarchy voids the correct-

[1] For more details on characterizing implementation steps using interpretation mappings, see our earlier paper [12].

[2] Our earlier paper [12] explains how to formalize structural descriptions of architectures as logical theories. Since structural descriptions are largely declarative, the process is quite straightforward.

ness guarantee provided by the restriction to verified transformations. Is there any way to allow the user to include such arbitrary steps in the development of the architecture hierarchy, while maintaining a correctness guarantee?

2 Proof-carrying architectures

Our solution to this problem is based on the notion of checking the correctness of steps in architecture hierarchy development. By *checking*, I mean automatically performing some calculation that shows the step is correct. Checking can be substantially simpler than verification, because it is focussed on a particular step. Verifying a transformation means showing that it *always* produces correct results, while check a transformation step means showing that a correct result was obtained *in one specific case*. Thus, checking entirely avoids the sometimes difficult problem of characterizing the preconditions required for the transformation to produce correct results [21].

Our initial approach to checking transformation steps was inspired work by George Necula and Peter Lee on compilers that generate proof-carrying code (PCC) [15]. Their basic idea is that, rather than attempting to prove the transformations performed by a compiler always produce code with certain desired properties, to generate a purported formal proof that the complied code has those properties as part of the code generation process. The purported proof can then be checked and, if it turns out to be a correct proof, it follows that the generated code has the desired properties. Thus, the emphasis is shifted from showing that compiler transformations are correct in general to checking that they produced correct results in individual cases.

The application of this idea to architectural transformation is straightforward. At some abstract level, the architectural description is proven to guarantee that the architecture has some desirable property, **C**. The interpretation mapping μ that sends abstract level sentences to their implementations can also be applied to the proof of **C**. If the image of the proof under the implementation mapping turns out to be a correct proof that the implementation has $\mu(\mathbf{C})$, then, of course, the implementation has $\mu(\mathbf{C})$. Checking the transformed proof can, therefore, provide the desired correctness guarantee.

3 An example: Secure distributed transaction processing

The idea of proof-carrying architectures can be illustrated by an example, based on our development of software architectures for secure distributed transaction processing (SDTP) [13]. These architectures extend X/Open's standard DTP architecture [23] by enforcing

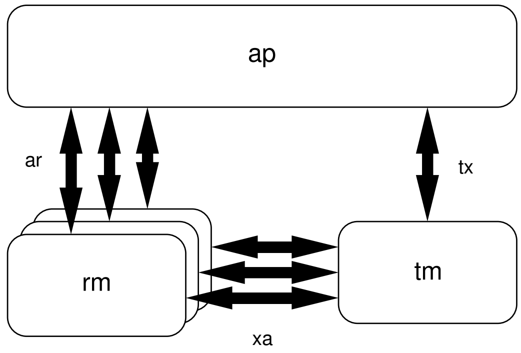

Figure 1: Abstract SDTP architecture — components linked by secure channels

a simple "no read up, no write down" security policy. The primary result of our development efforts is a hierarchy that links an extremely abstract architectural description, shown in Fig. 1, to three implementation-level descriptions written in a style that can be directly translated into a programming language such as Java using standard network programming constructs. The gap between the abstract SDTP architecture and each concrete SDTP architectures is filled by roughly two dozen — the exact number varies among the implementations — descriptions at intermediate levels of abstraction, linked in a chain by interpretation mappings. We are in the process of formally proving the implementation-level architectures secure by proving that the abstract description is secure, and proving that every interpretation mapping preserves security. One of the techniques that is being employed is showing that the interpretations incrementally transform the abstract-level security proof into implementation-level security proofs. The example below shows how the interpretation mapping associated with the first refinement step in all three chains transforms the abstract security proof into a slightly more concrete security proof.

3.1 The abstract SDTP architecture

Fig. 1 depicts the most abstract architecture for SDTP. The boxes are the components of the architecture: the Application (labeled "ap"), some number of Resource Managers (labeled "rm"), and a Transaction Manager (labeled "tm"). The components are linked by *secure channels*, indicated by the heavy doubleheaded arrows that make up the interfaces between the Application and Resource Managers, the Application and the Transaction Manager, and the Resource Managers and the Transaction Manager. Secure channels are a type of connector that enforce the security policy. In other words, secure channels will not carry classified data from a component to a component that lacks required clearances. To say that the system as a whole satisfies the security policy means that there is no flow of classified data to a component that lacks the required clearances.

3.2 An abstract proof of security

Informally, the security of the system follows almost immediately from the the fact that it employs only secure channels [13]. Not surprisingly, a textbook-style natural deduction proof [8, 10] of system security is quite simple.[3] Consider the dataflow from some given Resource Manager rm to the Application ap, for example. A proof of the formula

$$(\forall \mathsf{d} : \mathsf{Labeled\_Data})[\mathsf{Flows}(\mathsf{d}, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{label}(\mathsf{d}) \leq \mathsf{clearance}(\mathsf{ap})]$$

which says

> every labeled datum d that flows from rm to ap has a security label classifying it that is less than or equal to the clearance level of ap

from five axioms of the architectural theory is shown in Fig. 2. The five axioms say

1. every labeled datum d that flows from rm to ap is carried by secure_ar_channel from the output port rm's_ar_port to the input port of the port array ap's_ar_ports that is indexed by rm,
2. the input port of the port array ap's_ar_ports that is indexed by rm is a port of ap,
3. if secure channel c carries labeled datum d from output port x to input port y, then d's security label is less than or equal to the clearance level of y,
4. the clearance level of any port y of component a must be less than or equal to the clearance level of a, and
5. the ordering of security labels is transitive.

The first two axioms are facts about the particular architecture, the third axiom is the defining property of the secure channel subtype, the fourth and fifth axioms are general axioms of the security model.

3.3 A slightly more concrete SDTP architecture

The secure channels of abstract SDTP architecture can be implemented in terms of ordinary dataflow channels and additional components in a variety of ways, depending upon the security properties of the components [13].

[3] In this paper, I will use natural deduction, since that provides a familiar concrete representation of formal proofs. In our actual verifications that the SDTP hierarchy's interpretation steps preserve security, we are employing the PVS verification system [18].

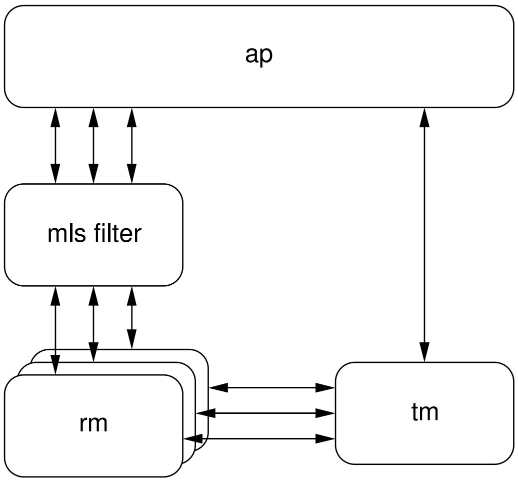

Figure 3: More concrete SDTP architecture — secure channels refined to ordinary channels, or ordinary channels plus security filter

The most interesting implementation is shown in Fig. 3, where the light doubleheaded arrows represent ordinary dataflow channels that do not enforce the security policy. This implementation is suited to the case where all of the resource managers are single-level, but not necessarily the same level. The security policy is enforced by a multi-level secure component that filters dataflow between the application and the resource managers: if passing a datum from a resource manager to the application would violate the security policy, the filter removes it from the stream.

The concrete architecture can be thought of as resulting from the abstract architecture by applying several transformations. For example, one transformation, the *Filter Introduction Transformation* (*FIT*), replaces secure channels between components that are not multilevel secure by ordinary dataflow channels and a component that enforces the security policy.

3.4 A slightly more concrete proof of security

Now it must be shown that, like the abstract SDTP architecture, the more concrete SDTP architecture has the desired security property. The two conventional approachs to establishing this result are

1. to directly prove that the more concrete architecture is secure, in much the same way the the abstract architecture was proven secure (perhaps using the abstract-level proof for heuristic guidance), and
2. to show that the Filter Introduction Transformation (FIT), and the other transformations that produce the more concrete architecture from the ab-

$\{1\}$	1.	$(\forall \mathsf{d} : \mathsf{Labeled\_Data})[\mathsf{Flows}(\mathsf{d}, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{Carries}(\mathsf{secure\_ar\_channel}, \mathsf{d}, \mathsf{rm's\_ar\_port}, \mathsf{ap's\_ar\_ports}(\mathsf{rm}))]$	Axiom describing specific architecture
$\{2\}$	2.	$\mathsf{Port\_Of}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}), \mathsf{ap})$	Axiom describing specific architecture
$\{3\}$	3.	$(\forall \mathsf{c} : \mathsf{Secure\_Channel})(\forall \mathsf{d} : \mathsf{Labeled\_Data})(\forall \mathsf{x} : \mathsf{Output\_Port})(\forall \mathsf{y} : \mathsf{Input\_Port})[\mathsf{Carries}(\mathsf{c}, \mathsf{d}, \mathsf{x}, \mathsf{y}) \supset \mathsf{label}(\mathsf{d}) \leq \mathsf{clearance}(\mathsf{y})]$	Axiom characterizing secure channels
$\{4\}$	4.	$(\forall \mathsf{a} : \mathsf{Component})(\forall \mathsf{y} : \mathsf{Input\_Port})[\mathsf{Port\_Of}(\mathsf{y}, \mathsf{a}) \supset \mathsf{clearance}(\mathsf{y}) \leq \mathsf{clearance}(\mathsf{a})]$	Axiom constraining port clearances
$\{5\}$	5.	$(\forall \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3 : \mathsf{Security\_Label})[\mathsf{s}_1 \leq \mathsf{s}_2 \wedge \mathsf{s}_2 \leq \mathsf{s}_3 \supset \mathsf{s}_1 \leq \mathsf{s}_3]$	Axiom specifying transitivity of security label ordering
$\{1\}$	6.	$\mathsf{Flows}(\mathsf{d}_0, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{Carries}(\mathsf{secure\_ar\_channel}, \mathsf{d}_0, \mathsf{rm's\_ar\_port}, \mathsf{ap's\_ar\_ports}(\mathsf{rm}))$	Universal instantiation (1)
$\{3\}$	7.	$\mathsf{Carries}(\mathsf{secure\_ar\_channel}, \mathsf{d}_0, \mathsf{rm's\_ar\_port}, \mathsf{ap's\_ar\_ports}(\mathsf{rm})) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}))$	Universal instantiation (3)
$\{1,3\}$	8.	$\mathsf{Flows}(\mathsf{d}_0, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}))$	Tautological consequence (6,7)
$\{4\}$	9.	$\mathsf{Port\_Of}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}), \mathsf{ap}) \supset \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \leq \mathsf{clearance}(\mathsf{ap})$	Universal instantiation (4)
$\{2,4\}$	10.	$\mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \leq \mathsf{clearance}(\mathsf{ap})$	Tautological consequence (2,4)
$\{5\}$	11.	$\mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \wedge \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \leq \mathsf{clearance}(\mathsf{ap}) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap})$	Universal instantiation (5)
$\{2,4,5\}$	12.	$\mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap})$	Tautological consequence (10,11)
$\{1,2,3,4,5\}$	13.	$\mathsf{Flows}(\mathsf{d}_0, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap})$	Tautological consequence (8,12)
$\{1,2,3,4,5\}$	14.	$(\forall \mathsf{d} : \mathsf{Labeled\_Data})[\mathsf{Flows}(\mathsf{d}, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{label}(\mathsf{d}) \leq \mathsf{clearance}(\mathsf{ap})]$	Universal generalization (13)

Figure 2: Formal proof that dataflow from rm to ap satisfies the security policy

stract architecture, always preserves the security propery.

The use of proof-carrying architectures provides a third alternative.

When transformation FIT is applied, it can be applied not only to the architectural description, but to the formal security proof of Fig. 2 as well. The result of applying FIT to this proof is shown in Fig. 4, where the implementation mapping μ associated with this application is determined as follows. A complete account of how first-order interpretation mappings are defined, and basic facts about them, can be found in the logic textbooks of Enderton [4] and Shoenfield [22], among others. [4] For present purposes, it is enough to know that

1. for every term $\mathbf{t}$ of the language of the abstract theory, $\mu(\mathbf{t})$ is a (possibly complex) term of the language of the more concrete theory,

2. for every predicate $\mathbf{F}$ of the language of the abstract theory, $\mu(\mathbf{F})$ is a (possibly complex) predicate of the language of the more concrete theory,

3. for every formula $\mathbf{A}$ of the language of the abstract theory,
$$\mu(\neg\mathbf{A}) = \neg\mu(\mathbf{A})$$
and similarly for the other connectors, and

4. for every formula $\mathbf{A}$ of the language of the abstract theory, every variable $\mathbf{x}$, and every type predicate $\mathbf{T}$ of the language of the abstract theory,
$$\mu((\forall\mathbf{x} : \mathbf{T})\,\mathbf{A}) = \mu(\forall\mathbf{x} : \mathbf{T})\,\mu(\mathbf{A})$$
where $\mu(\forall\mathbf{x} : \mathbf{T})$ is a sequence of universal quantifiers, and similarly for the other quantifiers.

TheCarries predicate

$$\mathsf{Carries}(\langle \textit{secure channel}\rangle,\ \langle \textit{datum}\rangle,\ \langle \textit{out port}\rangle,\ \langle \textit{in port}\rangle)$$

that is mentioned in formulas 1, 3, 6, and 7 of the abstract-level proof is mapped to a conjunction of the Carries, Passes, and Carries predicates,

$$\begin{aligned}&\mathsf{Carries}(\langle \textit{channel}\rangle,\ \langle \textit{datum}\rangle,\ \langle \textit{out port}\rangle,\ \langle \textit{filter in port}\rangle)\\ &\quad\wedge \mathsf{Passes}(\langle \textit{filter}\rangle,\ \langle \textit{datum}\rangle,\ \langle \textit{filter in port}\rangle,\ \langle \textit{filter out port}\rangle)\\ &\quad\wedge \mathsf{Carries}(\langle \textit{channel}\rangle,\ \langle \textit{datum}\rangle,\ \langle \textit{filter out port}\rangle,\ \langle \textit{in port}\rangle)\end{aligned}$$

This clause in the definition of μ says that a secure channel carrying a datum from some output port to some input port is implemented as a channel carrying the datum from the output port to some input port of a filter, passing the datum through the filter from the input port to some output port, and carrying the datum from output port of the filter to the input port.[5] This mapping is also applied to formula 3 in order to preserve the fact that formula 7 should follow from formula 3 by Universal Instantiation. The universal quantifier over secure channels in formula 3,

$$(\forall\ \langle \textit{secure channel variable}\rangle : \mathsf{Secure\_Channel})$$

is mapped by μ to to universal quantifiers over channels and a universal quantifier over MLS components,

$$\begin{aligned}&(\forall\ \langle \textit{to-filter channel variable}\rangle : \mathsf{Channel})\\ &\qquad(\forall\ \langle \textit{filter variable}\rangle : \mathsf{MLS\_Component})\\ &\qquad(\forall\ \langle \textit{from-filter channel variable}\rangle : \mathsf{Channel})\end{aligned}$$

It is easy to check that the result of applying the FIT interpretation mapping μ to the proof of security is a syntactically correct derivation of the desired security property from formulas that are images of axioms of the more abstract architectural theory. Mapping μ sends tautological consequence steps to correct tautological consequence steps, universal instantiation steps to correct universal instantiation steps, and universal generalization steps to correct universal generalization steps. So μ has indeed mapped the formal abstract-level security proof to a concrete-level security proof, but not necessarily a proof *from axioms of the concrete architectural theory.*

3.5 Completing the proof

The image of the first axiom under μ says that every labeled datum that flows from rm to ap is carried to the filter from rm, passed through the filter, and then carried to ap from the filter. Just as in the case of the first axiom, this is a fact about the particular architecture that is either an axiom of the concrete theory, or easily and automatically derivable from axioms of the concrete theory. The mapping μ leaves the second axiom unchanged. This will certainly be an axiom of the concrete theory, as well as the abstract theory. The image of the third axiom is a bit more complex. It states that the combination of the two channels and the filter enforces the security property. It is quite unlikely that this would be among the chosen axioms of the concrete level theory, since it is the filter alone, effectively, that

[4] Technically, I will make use of what are called n-dimensional interpretations [7, pp. 212]. But this is a reasonably straightforward generalization of the definition found in the cited textbooks.

[5] This mapping would not be appropriate to apply to every occurrence of the Carries predicate in every derivation, because some secure channels in the abstract architecture may not be replaced by a combination of two channels and a filter in the concrete architecture. However, formulas 1, 6, and 7 of the proof specifically refer to what secure_ar_channel carries, and this secure channel *is* being implemented by two channels and a filter, so I will use this simpler interpretation for purposes of the example.

$\{1\}$	1.	$(\forall \mathsf{d} : \mathsf{Labeled\_Data})[\mathsf{Flows}(\mathsf{d}, \mathsf{rm}, \mathsf{ap})$
		$\supset \mathsf{Carries}(\mathsf{rm\_to\_filter\_channel}, \mathsf{d}, \mathsf{rm's\_ar\_port}, \mathsf{filter\_in\_port}(\mathsf{rm}))]$
		$\wedge \mathsf{Passes}(\mathsf{mls\_filter}, \mathsf{d}, \mathsf{filter\_in\_port}(\mathsf{rm}), \mathsf{filter\_out\_port}(\mathsf{rm}))]$
		$\wedge \mathsf{Carries}(\mathsf{filter\_to\_ap\_channel}(\mathsf{rm}), \mathsf{d}, \mathsf{filter\_out\_port}(\mathsf{rm}), \mathsf{ap's\_ar\_ports}(\mathsf{rm}))]$
$\{2\}$	2.	$\mathsf{Port\_Of}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}), \mathsf{ap})$
$\{3\}$	3.	$(\forall \mathsf{c}_1 : \mathsf{Channel})(\forall \mathsf{f} : \mathsf{MLS\_Component})(\forall \mathsf{c}_2 : \mathsf{Channel})(\forall \mathsf{d} : \mathsf{Labeled\_Data})$
		$(\forall \mathsf{x}_1 : \mathsf{Output\_Port})(\forall \mathsf{x}_2 : \mathsf{Output\_Port})(\forall \mathsf{y}_1 : \mathsf{Input\_Port})(\forall \mathsf{y}_2 : \mathsf{Input\_Port})$
		$[\mathsf{Carries}(\mathsf{c}_1, \mathsf{d}, \mathsf{x}_1, \mathsf{y}_1) \wedge \mathsf{Passes}(\mathsf{f}, \mathsf{d}, \mathsf{y}_1, \mathsf{x}_2) \wedge \mathsf{Carries}(\mathsf{c}_2, \mathsf{d}, \mathsf{x}_2, \mathsf{y}_2)$
		$\supset \mathsf{label}(\mathsf{d}) \leq \mathsf{clearance}(\mathsf{y}_2)]$
$\{4\}$	4.	$(\forall \mathsf{a} : \mathsf{Component})(\forall \mathsf{y} : \mathsf{Input\_Port})[\mathsf{Port\_Of}(\mathsf{y}, \mathsf{a}) \supset \mathsf{clearance}(\mathsf{y}) \leq \mathsf{clearance}(\mathsf{a})]$
$\{5\}$	5.	$(\forall \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3 : \mathsf{Security\_Label})[\mathsf{s}_1 \leq \mathsf{s}_2 \wedge \mathsf{s}_2 \leq \mathsf{s}_3 \supset \mathsf{s}_1 \leq \mathsf{s}_3]$
$\{1\}$	6.	$\mathsf{Flows}(\mathsf{d}_0, \mathsf{rm}, \mathsf{ap})$
		$\supset \mathsf{Carries}(\mathsf{rm\_to\_filter\_channel}, \mathsf{d}_0, \mathsf{rm's\_ar\_port}, \mathsf{filter\_in\_port}(\mathsf{rm}))$
		$\wedge \mathsf{Passes}(\mathsf{mls\_filter}, \mathsf{d}_0, \mathsf{filter\_in\_port}(\mathsf{rm}), \mathsf{filter\_out\_port}(\mathsf{rm}))$
		$\wedge \mathsf{Carries}(\mathsf{filter\_to\_ap\_channel}(\mathsf{rm}), \mathsf{d}_0, \mathsf{filter\_out\_port}(\mathsf{rm}), \mathsf{ap's\_ar\_ports}(\mathsf{rm}))$
$\{3\}$	7.	$\mathsf{Carries}(\mathsf{rm\_to\_filter\_channel}, \mathsf{d}, \mathsf{rm's\_ar\_port}, \mathsf{filter\_in\_port}(\mathsf{rm})$
		$\wedge \mathsf{Passes}(\mathsf{mls\_filter}, \mathsf{d}_0, \mathsf{filter\_in\_port}(\mathsf{rm}), \mathsf{filter\_out\_port}(\mathsf{rm}))$
		$\wedge \mathsf{Carries}(\mathsf{filter\_to\_ap\_channel}(\mathsf{rm}), \mathsf{d}_0, \mathsf{filter\_out\_port}(\mathsf{rm}), \mathsf{ap's\_ar\_ports}(\mathsf{rm}))$
		$\supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}))$
$\{1,3\}$	8.	$\mathsf{Flows}(\mathsf{d}_0, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}))$
$\{4\}$	9.	$\mathsf{Port\_Of}(\mathsf{ap's\_ar\_ports}(\mathsf{rm}), \mathsf{ap}) \supset \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \leq \mathsf{clearance}(\mathsf{ap})$
$\{2,4\}$	10.	$\mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \leq \mathsf{clearance}(\mathsf{ap})$
$\{5\}$	11.	$\mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \wedge \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \leq \mathsf{clearance}(\mathsf{ap})$
		$\supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap})$
$\{2,4,5\}$	12.	$\mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap's\_ar\_ports}(\mathsf{rm})) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap})$
$\{1,2,3,4,5\}$	13.	$\mathsf{Flows}(\mathsf{d}_0, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{ap})$
$\{1,2,3,4,5\}$	14.	$(\forall \mathsf{d} : \mathsf{Labeled\_Data})[\mathsf{Flows}(\mathsf{d}, \mathsf{rm}, \mathsf{ap}) \supset \mathsf{label}(\mathsf{d}) \leq \mathsf{clearance}(\mathsf{ap})]$

Figure 4: Transformed formal proof that dataflow from rm to ap satisfies the security policy

$\{1\}$	1.	$(\forall \mathsf{c} : \mathsf{Channel})(\forall \mathsf{d} : \mathsf{Labeled\_Data})(\forall \mathsf{x} : \mathsf{Output\_Port})(\forall \mathsf{y} : \mathsf{Input\_Port})$ $[\mathsf{Carries}(\mathsf{c}, \mathsf{d}, \mathsf{x}, \mathsf{y}) \supset \mathsf{clearance}(\mathsf{x}) = \mathsf{clearance}(\mathsf{y})]$	Axiom specifying connection constraint imposed by security model
$\{2\}$	2.	$(\forall \mathsf{f} : \mathsf{MLS\_Component})(\forall \mathsf{d} : \mathsf{Labeled\_Data})(\forall \mathsf{y} : \mathsf{Input\_Port})(\forall \mathsf{x} : \mathsf{Output\_Port})$ $[\mathsf{Passes}(\mathsf{f}, \mathsf{d}, \mathsf{y}, \mathsf{x}) \supset \mathsf{label}(\mathsf{d}) \leq \mathsf{clearance}(\mathsf{x})]$	Axiom characterizing MLS components
$\{3\}$	3.	$(\forall \mathsf{x})(\forall \mathsf{y})(\forall \mathsf{z})[\mathsf{x} = \mathsf{y} \supset [\mathsf{z} \leq \mathsf{x} \equiv \mathsf{z} \leq \mathsf{y}]]$	Instance of identity axiom schema
$\{1\}$	4.	$\mathsf{Carries}(\mathsf{c}_2, \mathsf{d}_0, \mathsf{x}_2, \mathsf{y}_2) \supset \mathsf{clearance}(\mathsf{x}_2) = \mathsf{clearance}(\mathsf{y}_2)$	Universal instantiation (1)
$\{2\}$	5.	$\mathsf{Passes}(\mathsf{f}_0, \mathsf{d}_0, \mathsf{y}_1, \mathsf{x}_2) \supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{x}_2)$	Universal instantiation (2)
$\{1,2\}$	6.	$\mathsf{Carries}(\mathsf{c}_1, \mathsf{d}_0, \mathsf{x}_1, \mathsf{y}_1) \wedge \mathsf{Passes}(\mathsf{f}_0, \mathsf{d}_0, \mathsf{y}_1, \mathsf{x}_2) \wedge \mathsf{Carries}(\mathsf{c}_2, \mathsf{d}_0, \mathsf{x}_2, \mathsf{y}_2)$ $\supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{x}_2) \wedge \mathsf{clearance}(\mathsf{x}_2) = \mathsf{clearance}(\mathsf{y}_2)$	Tautological consequence (3,4)
$\{3\}$	7.	$\mathsf{clearance}(\mathsf{x}_2) = \mathsf{clearance}(\mathsf{y}_2) \supset [\mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{x}_2) \equiv \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{y}_2)$	Universal instantiation (3)
$\{1,2,3\}$	8.	$\mathsf{Carries}(\mathsf{c}_1, \mathsf{d}_0, \mathsf{x}_1, \mathsf{y}_1) \wedge \mathsf{Passes}(\mathsf{f}_0, \mathsf{d}_0, \mathsf{y}_1, \mathsf{x}_2) \wedge \mathsf{Carries}(\mathsf{c}_2, \mathsf{d}_0, \mathsf{x}_2, \mathsf{y}_2)$ $\supset \mathsf{label}(\mathsf{d}_0) \leq \mathsf{clearance}(\mathsf{y}_2)$	Tautological consequence (6,7)
$\{1,2,3\}$	9.	$(\forall \mathsf{c}_1 : \mathsf{Channel})(\forall \mathsf{f} : \mathsf{MLS\_Component})(\forall \mathsf{c}_2 : \mathsf{Channel})(\forall \mathsf{d} : \mathsf{Labeled\_Data})$ $(\forall \mathsf{x}_1 : \mathsf{Output\_Port})(\forall \mathsf{x}_2 : \mathsf{Output\_Port})(\forall \mathsf{y}_1 : \mathsf{Input\_Port})(\forall \mathsf{y}_2 : \mathsf{Input\_Port})$ $[\mathsf{Carries}(\mathsf{c}_1, \mathsf{d}, \mathsf{x}_1, \mathsf{y}_1) \wedge \mathsf{Passes}(\mathsf{f}, \mathsf{d}, \mathsf{y}_1, \mathsf{x}_2) \wedge \mathsf{Carries}(\mathsf{c}_2, \mathsf{d}, \mathsf{x}_2, \mathsf{y}_2)$ $\supset \mathsf{label}(\mathsf{d}) \leq \mathsf{clearance}(\mathsf{y}_2)]$	Universal generalization (8)

Figure 5: Proof of image of abstract-level formula 3 under μ from axioms of concrete theory

is enforcing security. Still, it is easy to see that this formula must be a consequence of axioms of the concrete theory: the security model requires that channels which do not enforce security can only connect ports with matching clearances, and one of the defining properties of an MLS component is that it only supplies data at an output port if the classification of the data is less than ot equal to the clearance of the port. A formalization of this proof from particular axioms we use in the SDTP security verification is shown in Fig. 5. Discovery of this proof is easy. The form of the desired conclusion — a conjunction of conditions on $\mathsf{Carries}$ and Passes in the antecedent, and the comparison of label to clearance in the consequent — immediately suggests the use of the axioms on lines 1 and 2 of the proof. So it should be quite plausible that the proof can be discovered without human intervention by the transformation system. The interpretation mapping μ does not affect the images of the remaining two axioms; they remain general axioms of the security model. So, by combining the proof in Fig. 5 with the proof in Fig. 4, we obtain a proof of the security property from axioms of the concrete theory. Moreover, this proof is recognizably a formalization of our informal argument [13, p. 89] that the concrete architecture satisfies the security policy.

4 Generalizing from the example

The idea of using the architectural transformation to transform the proof that the more abstract architecture has a desired property into a proof that the more concrete architecture has the property worked well for this rather simple, but real-world, example. Is there any reason to believe that it will work equally well in other cases?

Recall that the standard criterion for correctness of a implementation mapping μ of an abstract logical theory T_1 in a more concrete theory T_2 is that μ must interpret T_1 in T_2, i.e., it must be the case that, for every formula $\mathbf{A}$ in the language of T_1,

$$T_1 \vdash \mathbf{A} \quad \Longrightarrow \quad T_2 \vdash \mu(\mathbf{A})$$

If μ interprets T_1 in T_2, an easy inductive argument shows that μ maps formal proofs from T_1 to formal proofs from $\mu[T_1]$ that can be extended to proofs from T_2. If $\mathbf{A}$ is an axiom of T_1, then, since μ is a theory interpretation, $\mu(\mathbf{A})$ is derivable from T_2. Because μ is defined so that connectives pass through it, μ maps tautological consequence steps to tautological consequence steps. Similarly, μ maps universal instantiation and universal generalization steps to universal instantiation and generalization steps, respectively. Thus, μ maps formal proofs from abstract axioms to formal proofs from images of abstract axioms, and images of abstract axioms

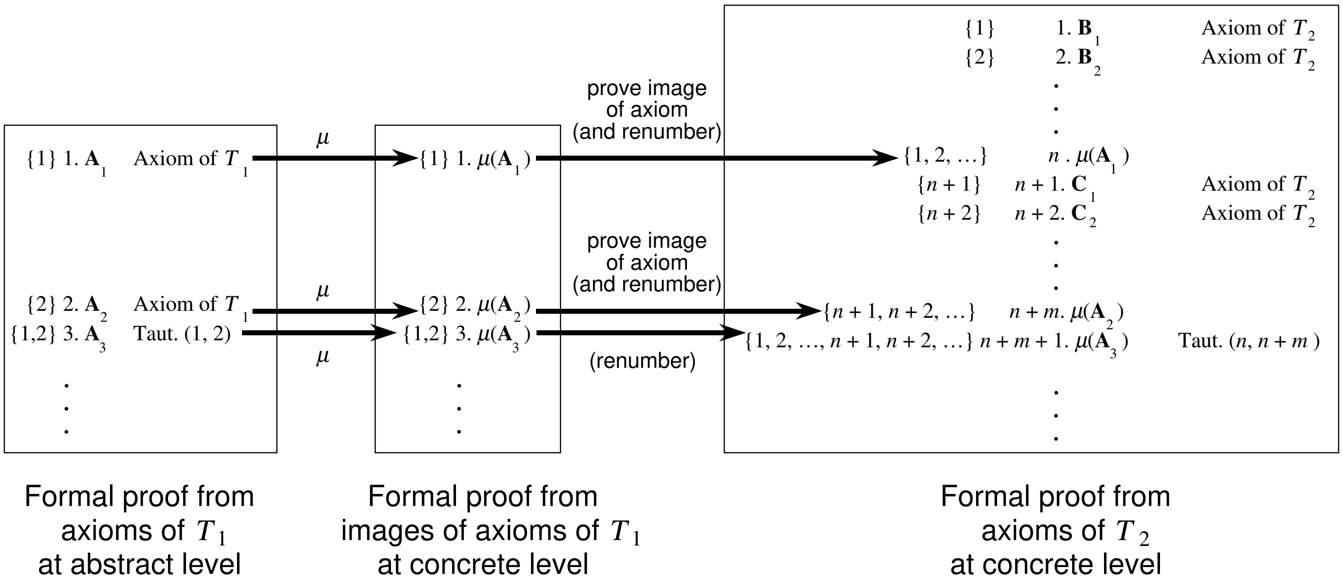

Figure 6: Interpretation of formal proofs

can always be proved from concrete axioms, as shown in Fig. 6.

So, if an architectural transformation step is correct, in the standard sense, the corresponding interpretation mapping will map formal proofs to formal proofs containing gaps that can be filled. A fortiori, an abstract-level formal proof of some particular property of interest — say, satisfaction of a security policy — will be mapped to a proof that the implementation also has (the implementation-level analogue of) the property. Since the replacement of the secure channel from `rm` to `ap` by a pair of channels and a filter is evidently correct, it is not surprising that the FIT mapping sends the abstract-level security proof to a concrete-level security proof.

It follows that the proof-carrying architecture approach allows the architect to perform arbitrary correct transformations when implementing an abstract architecture, provided the transformation system that supports the approach is clever enough to find the proofs of images of axioms. The question remains: In general, how hard is it to discover these proofs? In our experience, it invariably quite easy, because we deal with refinement patterns that make only small changes in representation of the architecture. Indeed, the example in Fig. 5 is representative of the complexity of most of these proofs. At lower levels in the SDTP hierarchy, there are more gaps to be filled in — because lower-level architectural theories are more complex, and proofs are based on a larger number of axioms — but the size of the gaps is about the same. We are confident that considerable automated support for finding proofs to fill the gaps can be provided.

Finally, it should be noted that incorrect transformations that happen to preserve the proof of the property of interest will also be judged acceptable on the proof-carrying architectures approach. Therefore, it is well-suited to the case where the focus is on obtaining an implementation with some particular desirable property — i.e., when a weaker-than-usual correctness criterion is adequate — and placing minimal constraints on the architect's implementation options is preferred, as is the case in SDTP.

5 Related work

Although there is a large and growing literature on formal software transformation, nearly all of it is oriented toward maintaining functional correctness, rather than system structure. Similarly, there is a large body of literature on architectural refinement and composition, nearly all of it employing semiformal representation and analysis techniques, at best. The comparatively few papers on formal refinement of architectural structure include Broy's work on component refinement [3], Brinksma, et al.'s, work on connector refinement [2], Philipps and Rumpe's recent work on refinement of information flow architectures [19], and the work described in our own earlier papers [11, 12, 13, 20, 21]. Also closely related is work by Garlan's group [1], Luckham's group [9], and Moriconi and Qian [11] on formally representing the semantics of connectors and relating semantic models at different levels of abstraction. But, the emphasis in all these cases has always been on verification of general refinement patterns, rather than checking particular steps.

Necula and Lee's work on proof-carrying code and its applications [15, 16, 17] introduced the notion of replacing verification by checking in the context of compilation. The work described in this paper can be viewed as generalizing their ideas about code refinement transformations to architectural transformations, both refinements and abstractions.

6 Conclusions

Transformational development of architectures can guarantee that implementations are correct by restricting the architect to a stock of verified transformations. But such a correctness guarantee is quite brittle, since use of a single non-verified transformation voids it. Moreover, if many transformations are used, and the verification of each is difficult, then confidence in the correctness of the implementation may be less than desired. Checking particular refinement steps offers a way of allowing the architect greater freedom, and of achieving higher levels of confidence that the implemented architecture has the desired properties.

Our initial approach to checking, based on the idea of proof-carrying architectures, is especially well suited to the case where the main requirement is high confidence that the implementation has some specific property. The property is shown to hold at some abstract level, and every refinement is produced by application of a transformation known to preserve the property, or is checked for correctness by making sure that the transformation preserves the proof of the desired property, or both.

The main limitation of this first approach to checking is that properties are checked one-at-a-time. We are exploring other approaches to checking that allow an entire class of properties to be checked at once. One that seems particularly promising is based on the idea of applying the simplified technique for proving implementation mapping correctness [20] to development steps at architecture definition-time. This complementary approach to checking will allow the correctness of steps to be checked, relative to our strong correctness criterion, rather than checking one or a few properties of interest. But it can be applied only to complete architectural descriptions of single structures, not to descriptions of varied families of architectural structures. The proof-checking architectures approach applies equally well to descriptions of single structures and descriptions of families.

As mentioned above, our preliminary experiments with proof-carrying architecture are being performed with the PVS verification system [18]. Improved support for working with proof-carrying architectures, including automated discovery of the gap-filling proofs, is being implemented as part of the Xform[6] system, an enhanced version of our present architectural correctness checking toolset. *Xform*, pronounced *transform*, is a recursive acronym for "*Xform, for orderly reification*[7] *and maintenance.*" Xform will support transformational development and maintenance of architectural descriptions written in languages such as SADL [14] and ACME [6].

[6] *Xform* is a common mathematical shorthand for *transform*.

[7] *To reify* means to make actual. Thus, reification of software architectures is the process of turning them into actual implementations.

REFERENCES

[1] ABOWD, G., ALLEN, R., AND GARLAN, D. Formalizing style to understand descriptions of software architecture. Tech. Rep. CMU-CS-95-111, School of Computer Science, Carnegie Mellon University, January 1995.

[2] BRINKSMA, E., JONSSON, B., AND ORAVA, F. Refining interfaces of communicating systems. In *Proceedings of TAPSOFT '91* (1991), S. Abramsky and T. S. E. Maibaum, Eds., Springer-Verlag, pp. 71–80.

[3] BROY, M. Compositional refinement of interactive systems. Tech. Rep. No. 89, Digital Systems Research Center, Palo Alto, July 1992.

[4] ENDERTON, H. B. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[5] GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. Exploiting style in architectural design environments. In *Proceedings 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94)* (1994), ACM Press, pp. 179–185.

[6] GARLAN, D., MONROE, R. T., AND WILE, D. Acme: An archiectural description interchange language. In *Proceedings of CASCON '97* (November 1997). Available at `http://www.cs.cmu.edu/afs/cs/project/abel/www/acme-web/v3.0/white-paper-v3.0/white-paper.html`.

[7] HODGES, W. *Model Theory.* Cambridge University Press, 1993.

[8] LEMMON, E. J. *Beginning Logic*, second ed. Chapman and Hall, 1987.

[9] LUCKHAM, D. C., AUGUSTIN, L. M., KENNEY, J. J., VERA, J., BRYAN, D., AND MANN, W. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering 21*, 4 (April 1995), 314–335.

[10] MATES, B. *Elementary Logic*, second ed. Oxford University Press, 1972.

[11] MORICONI, M., AND QIAN, X. Correctness and composition of software architectures. In *Proceedings 2nd ACM Symposium on Foundations of Software Engineering (SIGSOFT '94)* (1994), ACM Press, pp. 164–174.

[12] MORICONI, M., QIAN, X., AND RIEMENSCHNEIDER, R. A. Correct architecture refinement. *IEEE Transactions on Software Engineering 21*, 4 (April 1995), 356–372. Available at `http://www.csl.sri.com/sadl/tse95.ps.gz`.

[13] Moriconi, M., Qian, X., Riemenschneider, R. A., and Gong, L. Secure software architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (May 1997), pp. 84–93. Available at http://www.csl.sri.com/sadl/sp97.ps.gz.

[14] Moriconi, M., and Riemenschneider, R. A. Introduction to Sadl 1.0: A language for specifying software architecture hierarchies. Tech. Rep. SRI-CSL-97-01, Computer Science Laboratory, SRI International, March 1997. Available at http://www.csl.sri.com/sadl/sadl-intro.ps.gz.

[15] Necula, G. C., and Lee, P. The design and implementation of a certifying compiler. Submitted to PLDI '98. Available at http://www.cs.cmu.edu/~necula/pldi98.ps.gz.

[16] Necula, G. C., and Lee, P. Proof-carrying code. Tech. Rep. CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, November 1996. Available at http://www.cs.cmu.edu/~necula/tr96-165.ps.gz.

[17] Necula, G. C., and Lee, P. Efficient representation and validation of logical proofs. Tech. Rep. CMU-CS-97-172, School of Computer Science, Carnegie Mellon University, October 1997. Available at http://www.cs.cmu.edu/~necula/tr97-172.ps.gz.

[18] Owre, S., Rushby, J. M., and Shankar, N. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)* (Saratoga, NY, June 1992), D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 748–752.

[19] Philipps, J., and Rumpe, B. Refinement of information flow architectures. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM '97)* (November 1997), pp. 203–212. Available at http://www4.informatik.tu-muenchen.de/papers/icfem_rumpe_1997_Publication.html.

[20] Riemenschneider, R. A. A simplified method for establishing the correctness of architectural refinements. SRI CSL Dependable System Archiecture Group, Working Paper DSA-97-02. Available at http://www.csl.sri.com/sadl/simplified.ps.gz., November 1997.

[21] Riemenschneider, R. A. Correct transformation rules for incremental development of architecture hierarchies. SRI CSL Dependable System Archiecture Group, Working Paper DSA-98-01. Available at http://www.csl.sri.com/sadl/incremental.ps.gz., February 1998.

[22] Shoenfield, J. R. *Mathematical Logic*. Addison-Wesley, 1967.

[23] X/Open Company. *Distributed Transaction Processing: Reference Model*. Apex Plaza, Forbury Road, Reading, Berkshire RG1 1AX, U.K., November 1993.