A Formalization of Software Architecture

John Herbert, Bruno Dutertre, Robert Riemenschneider, and Victoria Stavridou

Dependable System Architecture Group
System Design Laboratory
SRI International
Menlo Park CA 94025
USA
{herbert, rar, victoria, bruno}@sdl.sri.com
http://www.sdl.sri.com/dsa/

Abstract. Software architecture addresses the high level specification, design and analysis of software systems. Formal models can provide essential underpinning for architectural description languages (ADLs), and formal techniques can play an important role in analysis.
While formal models and formal analysis may always enhance conventional notations and methods, they are of greatest benefit when they employ tractable models and efficient, mechanisable techniques. The novelty in our work has been in the effort to find and mechanise a general semantic framework for software architectures that can provide tractable models and support architectural formal analysis.
The resultant semantic framework is a layered one: the core is a simple model of the elements and topology, which provides the basis for general architectural theorems and proof techniques; the structural core is augmented by semantic layers representing the semantics of relevant properties of the design.
The model has been implemented in the higher-order logic proof tool PVS, and has been used in correctness proofs during a case study of a distributed transaction protocol.

1 Introduction

Software architecture research has resulted in a range of formalisms for modelling architectures including [16, 11, 2, 7, 8, 6]. The formal models of software architecture make possible formal analysis. Formal analysis is especially important for software architectures since operational models, amenable to conventional techniques, may be absent at this abstract level.

Formal verification of real-world designs is difficult. Apart from the use of model-checkers for hardware designs, formal verification is usually a tour de force effort. The success of model-checking relies on problems where the state space can (in effect) be enumerated and exhaustively checked. The abstract levels of design addressed by software architectures require more general proof methods such as induction.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 116–133, 1999.

For formal analysis of real-world designs to be effective one must have tool support, and the tools must provide efficient proof procedures. The kind of analysis, and consequently tool support, depends on the choice of underlying semantic model. The following describes our exploration of two approaches to the embedding of semantics.

1.1 A Precise Semantic Embedding

SADL (Structural Architectural Description Language) [11] is an example of a current architectural description language (ADL). Like similar ADLs, it has a rich type system and allows one to describe designs at various levels of abstraction. Appendix A presents a high level description in SADL of a software architecture. It describes a set of components and their connections; the configuration is illustrated in figure 2.

Aspects of SADL were inspired by the logic of PVS [12] so it is not surprising that a very precise semantic model can be constructed in PVS. Here is a fragment of the SADL description in Appendix A, followed by a translation into PVS. It illustrates the declaration of types, component types, an instance of a component and a connection.

```
...
tx_commands, tx_responses: TYPE
...
ap:  TYPE <= Function [ap_in1: ar_resources, ap_in2: tx_responses
                       -> ap_out1: ar_requests, ap_out2: tx_commands]
...
the_ap:  ap
...
ar_1: CONNECTION =
        (EXISTS c: Channel<ar_requests>)
           Connects(c, the_ap.ap_out1, the_rms.rm_in1)
...
```

```
...
tx_commands, tx_responses: TYPE
...
ap: TYPE =  Fun[[# ap_in1: ar_resources, ap_in2: tx_responses #],
               [# ap_out1: ar_requests, ap_out2: tx_commands #]]
...
the_ap: VAR  ap
...
ar_1: AXIOM
EXISTS (c: Channel[ar_requests]):
Connects(c, ap_out1(rng(the_ap)), rm_in1(dom(the_rms)))
...
```

There are a number of good points to note about this PVS translation: the syntax closely follows the SADL thus offering the advantage of transparency; the expressiveness of PVS, including the rich type system, matches that of SADL.

In principle, we have an excellent embedding and indeed we can (and have) done proofs of correctness based on this translation.

What is wrong with this embedding? The main problem is that the embedding is too precise in that each architecture defines its own types, has its own type structure, and is unique rather than an instance of a generic model of an architecture. Thus, while it is possible to do formal analysis of each individual architecture, it is not possible to derive general theorems that relate to many configurations or to easily implement general analysis techniques. The issue is not one of semantics; rather it is one of pragmatics, efficiency and generality.

1.2 Layered Embedding

The initial experiments in mechanising SADL suggest that a very precise embedding of semantics lacks generality and may not support efficient analysis. The solution is to adopt a lightweight approach using a layered semantic model. Rather than constructing a monolithic semantics covering the complete language, one provides a generic semantic framework. The layered semantic framework consists of:

- a core structural semantic model representing the design elements and their interconnection
- semantic layers representing behavioural or non-behavioural aspects of the design

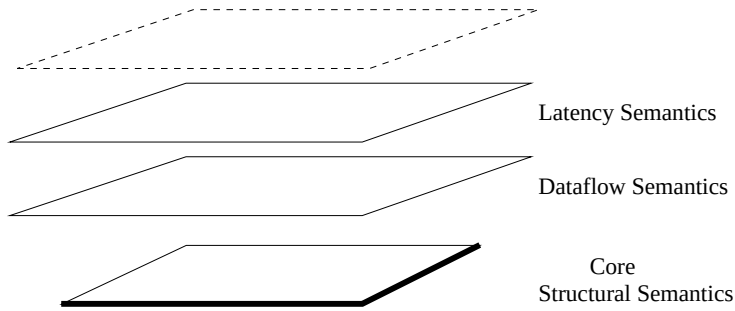

Fig. 1. Semantic layers

A design described in SADL is not translated to a single precise semantics, instead it is abstracted into the core structural semantics plus appropriate semantic layers [1]. A software architecture is a combination of design topology and properties. The layered semantic approach encourages a separation of concerns:

[1] The use of the term "layers" does not mean that a semantics must be built up sequentially. Independent semantic layers may augment the core structural description in parallel.

the core describes the components, connectors and their interconnection; each semantic layer describes related semantic properties. Figure 1 shows an example software design which is decomposed into the core structural model, and layers dealing with latency and dataflow.

1.3 Supporting Different ADLs

While originally developed for SADL, the layered semantic model is a generic solution applicable much more widely. This follows the motivation of ACME [2, 4] where the ACME notation provides a method of interchange of common aspects of designs. Our focus is complementary: on semantic integration rather than syntax-based interchange, but similar to ACME in that the core is a common description of structure and the other design information is structured — using the property languages for ACME and the semantic layers for our model.

Each semantic layer represents a model of some property of concern such as dataflow or communication bandwidth. The semantic layers are language independent and provide a model for semantic integration. Descriptions of two components in different ADLs may each result in statements at a common semantic layer and can therefore be combined in architectural analysis at this layer.

Our model avoids the need for full semantic translation between ADLs and provides the more feasible approach of translation into a common framework. Each ADL description is mapped onto the layered semantics. A heterogeneous design description employing various ADLs can therefore be abstracted into:

- the core language-independent elements and topology
- semantic information from the various descriptions structured into layers

Some layers may be relevant to just one language; layers representing standard concerns may be relevant to all languages. A shared semantic layer enables a full architectural analysis of the associated property across the heterogeneous components of the design.

2 Formal Model

An example of a simple software architecture structure[2] is given in figure 2. The core structural semantic model needs to represent components, the ports of the components, and the connections which relate ports of components. The model must support the description of families of elements as well as individual elements. This is done by defining a general type of components or connections; the actual components and connections of a design are then instances of this type.

The model has been formalised as a number of theories in PVS. The higher-order logic with dependent types of PVS provides an expressive language for

[2] A description of this architecture is provided in section 4.1

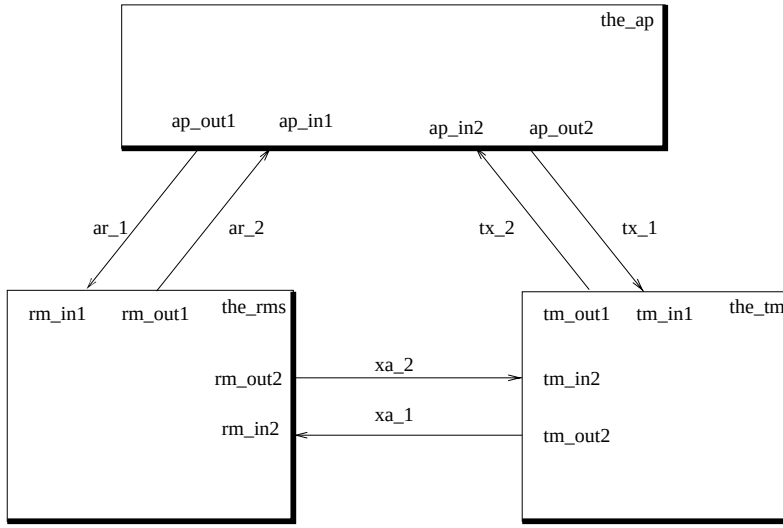

Fig. 2. Software structure: SDTP architecture example

embedding semantics. Formal analysis based on the semantics benefits from the high degree of automation in PVS.
(The PVS examples use the syntax of the prover: `FORALL, EXISTS, IMPLIES, AND` and `IFF` for the non-ascii $\forall, \exists, \supset, \bigwedge$ and $\Leftrightarrow$.
`a:bool` shows type decoration; `comp:(INST(comp_ty))` shows type dependency. `bool` is the type of booleans in PVS and appears in definitions as the domain when predicates are defined.
`(: :)` can denote a list and also a set via the implicit list2set coercion. It is used here to repesent a set, for example `(:output1,output2:)`.)

2.1 Components

Component families are declared using `MK_COMP`. This is defined as follows in PVS:

```
comp_ty: VAR COMP_TY
comp: VAR COMP
inputs: VAR setof[INPUT]
outputs: VAR setof[OUTPUT]
MK_COMP(comp_ty,inputs,outputs): bool =
   FORALL(comp:(INST(comp_ty))):
   (FORALL ip:
     INPUT_OF(comp)(ip) IFF member(ip,inputs)) AND
   (FORALL op:
     OUTPUT_OF(comp)(op) IFF member(op,outputs))
```

COMP_TY is a PVS type, an instance of which represents a component type; COMP is a PVS type, an instance of which represents an individual component. INPUT and OUTPUT are PVS types representing inputs and outputs.

For a family of component types comp_ty the definition of MK_COMP simply states that the only inputs and outputs of any component instance are those given by sets inputs and outputs.

The interaction points of component types and components are referred to as *inputs* and *outputs*. The fully qualified names for component instances are called *input ports* and *output ports* and are given by expressions such as: port(the_ap,ap_in1) and port(the_ap,ap_out2), where the_ap is a component instance and ap_in1 and ap_out2 are declared as input and output, respectively, of the component type.

Instances of component families are declared using an uninterpreted constant[3] INST. Components may also have constraints which can be stated by predicates on the input and outputs. The following example: declares a type box_ty to have one input and two outputs; makes an assertion that the values of one output depend on the input values while the values of the other output do not; declares the_box to be an instance of box_ty.

```
MK_COMP(box_ty,(:input:),(:output1,output2:))
FORALL(box:(INST(box_ty))):
   DependsOn(port(box,output1),port(box,input)) AND
   NOT(DependsOn(port(box,output2),port(box,input)))
INST(box_ty)(the_box)
```

2.2 Connections

A general type of connections, CONN_TY, and an uninterpreted constant CONNECTS, describing an unconstrained connection between an output port and input port, are declared.
A one-to-one connection is defined by:

```
CONNECTS11(conn,port(c1,output),port(c2,input)) =
(FORALL p1, p2:
 CONNECTS(conn,p1,p2) IFF (p1=port(c1,output)) AND (p2=port(c2,input)))
 AND
(FORALL conn1, p2:
 CONNECTS(conn1,port(c1,output),p2) IFF (conn1=conn)) AND
(FORALL conn1, p1:
 CONNECTS(conn1,p1,port(c2,input)) IFF (conn1=conn))
```

This states that: the only ports connected by conn are those given by the arguments; the only connection from the given output port is conn; the only connection to the given input port is conn.

[3] An uninterpreted constant is one with a type signature but without a defining axiom.

2.3 The Semantic Layers

A software architecture is not only a structure but a structure plus the behavioural and non-functional properties of the elements of the architecture. Semantic layers are a mechanism for partitioning the non-structural information. A layer provides a semantics for a concern of the designer.

Layering supports the separation of concerns: the core model provides the underlying structural semantics; the higher semantic layers represent different aspects of the design. The semantic layer may be completely independent or may sometimes be based on the underlying structure. Examples of layered semantic properties are latency between ports and dataflow.
The latency [4] semantic layer is based on assertions about pairs of ports; these are independent of the underlying structural semantics:

```
LATENCY(portA,portB,N)
```

The dataflow semantic layer is derived from the underlying structural semantics:

```
DirectFlow(c1,c2): bool =
  EXISTS conn, output, input:
  CONNECTS(conn,port(c1,output),port(c2,input));
```

2.4 Abstraction

Abstraction is required for mapping a design into the semantic framework. The result of the abstraction is a representation of the architectural topology in the core structural model and a representation of the behavioural and non-behavioural aspects of the design in the semantic layer(s). Mapping an ADL design to the core structural model is straightforward. However a certain collation of information may be required. In SADL for example, components are declared with explicit inputs and outputs but ignoring inter-component procedure calls. The ability to pass and return values of these procedure calls means they can act as communication ports. These implicit input and output ports must be made explicit in the mapping to the core model.

An important abstraction is type abstraction where the precise structured types of components, ports and connectors are ignored in the core model. Components have untyped input and output ports as points of interaction. The generic structure given by type abstraction supports the use of general theorems and techniques. The dropping of type information may be justified by noting that the original description is type correct and the structural connections are unchanged by the abstraction. One may need some type information for the semantic layers. This may be encoded as a predicate, for example `CARRIES(conn,values)`.

[4] Latency is modelled as a three place predicate relating two ports and a latency measure. The latency of the computations taking values on a certain port to values on another is thus described by a single number.

Various other abstractions may be useful in mapping from a design to the generic model. A comprehensive overview of abstractions for software architectures is given in [15]; our focus is on low level abstraction mappings. For example, structural abstraction may be used to abstract parameterised, replicated components to a finite representation. A group working paper [5] describes a number of these abstraction techniques.

While our original motivation was to provide a general tractable basis for analysis of SADL designs, the semantic framework developed can provide a basis for analysis of designs described in other ADLs or a design described using a mixture of ADLs. Abstraction functions appropriate to individual ADLs can be used to map into the semantic framework. The following example shows how one might abstract a Rapide text into the semantic framework.

Example: Abstraction of Rapide Rapide is an ADL developed at Stanford [7]. The semantics of Rapide is stated in terms of posets of events, and one of the main benefits of the language is its use in simulation. The following is a simple description of a component in Rapide:

```
type Resource is interface
   public action Receive(Msg : String);
   extern action Results(Msg : String);
constraint
   match
      ((?S in String) (Receive(?S) -> Results(?S)))^(*~);
end Resource;
the_resource: Resource;
```

The semantics of the component may be understood in Rapide in terms of the constraints on the sets of external events on ports `Receive` and `Results`. The relationship between the events may be stated precisely in Rapide to provide an accurate simulation model. For the purposes of formal analysis this behaviour may be abstracted to the property of concern being analysed.

If we are only interested in possible data dependency between ports then the detailed relationship between events can be ignored, and the abstraction can yield the fact that the `Results` port can depend on the `Receive` port.

```
MK_COMP(Resource,(:Receive:),(:Results:)) AND
INST(Resource)(the_resource) AND
FORALL r: (INST(Resource)):
   PortDependency(port(r,Receive),port(r,Results))
```

3 Architectural Analysis

The goal of the simple layered semantic model is to provide a lightweight embedding supporting effective *architectural* proof. Architectural proof emphasises proof that is based on the underlying architectural structure augmented with the semantic layer of concern.

Standard proof techniques are based on the logical syntax of the description rather than on the underlying architectural structure of the design. Thus, for example, a compound design may be required to satisfy the conjunction of the predicates describing its components. Standard manipulation of the logical operators quickly removes any reflection of the design architecture.

The abstraction of designs to the core structural semantics and semantic layers means that generic techniques based on the structure can underpin analysis at the semantic layer and can be applied to a wide range of designs. The following are some examples of architectural proof techniques.

3.1 Combining Architectural Measures

In the core model all components, ports and connectors are generic and so it is possible to formulate general techniques of combining measurements for components, ports, and connectors applicable to all designs.

An example is a general method of deriving from a measurement of a property relating port A and B, and another measurement relating ports B and C, the measurement of the property as it relates port A and C. A higher-order predicate `PROP` is declared and an axiom defining its semantics is introduced:

```
PROP(fn)(portA,portB,X) AND
PROP(fn)(portB,portC,Y) IMPLIES
  PROP(fn)(portA,portC,fn(X,Y))
```

i.e. if a property with a combining function `fn` has a measurement `X` between `portA` and `portB` and measurement `Y` between `portB` and `portC` then the property between `portA` and `portC` is computed as `fn(X,Y)`

Depending on the property, the user can choose an appropriate function, `fn`, to calculate the resultant property measure. Thus, latency between ports is additive so `fn` is a λ-term returning the sum of its arguments:

```
LATENCY(portA,portB,N) :bool =
  PROP(LAMBDA (a,b:int): a+b)(portA,portB,N)
```

Given a property measured by rationals and multiplicative, for example probability of dropping data, then the appropriate definition would be:

```
PROB_LOSS(portA,portB,NR) :bool =
  PROP(LAMBDA (a,b:rat): a*b)(portA,portB,NR)
```

By implementing the general pattern of inter-port measurement using `PROP` one can cover a number of properties.

3.2 Transitive Closure

Certain architectural two-place predicates have a common property, namely, they are transitive, and for these a general method of architectural analysis can be used: computation of the transitive closure. Given a transitive relation describing properties of individual elements, the transitive closure of the relation describes the property for the complete architecture.

The computation of transitive closure demands a finite structure but non-finite replicated structures can be abstracted to provide a finite structure over which one can compute the transitive closure [5]. For a domain of n objects then the maximum size of the transitive closure n^2.

A typical transitive property of software architectures is dataflow between components. The flow between adjacent components may be described by a set of predicates: `(:Flow(A,B), Flow(B,C), Flow(B,D), Flow(D,B):)`

It is straightforward to compute the transitive closure of the *Flow* relation over this domain. The result describes all possible flows, direct and indirect, resulting from the architectural structure:

```
(:Flow(A,B), Flow(B,C), Flow(B,D), Flow(D,B), Flow(A,C),
  Flow(A,D), Flow(B,B), Flow(D,C), Flow(D,D):)
```

4 Application Example

Previous sections have described the formal model and support for architectural analysis. Applications drive past and future development of the semantic framework and mechanised analysis techniques. Our main example is a particular architecture, namely the SDTP reference architecture. This section illustrates the use of semantic layers and architectural proof through the formal analysis of an abstract level of this architecture.

4.1 SDTP Reference Architecture

The SDTP (Secure Distributed Transaction Protocol) architecture [10] implements the X/Open DTP (Distributed Transaction Processing) standard [17] enhanced to support multilevel security. Figure 3 provides a diagram of an abstract level of the reference architecture.

The architecture consists of an application, `the_ap`, a transaction manager, `the_tm`, and resource managers (incorporating resources), `the_rms`. The general operation of the protocol is that the application contacts the transaction manager to initiate the transaction; the transaction manager then sets up the transaction, returning information to the application and contacting the resource managers to prepare appropriate resources for the transaction; the application then contacts the resource managers and conducts the transaction; further calls from the application to the transaction manager (and then from transaction manager to resource managers) can commit or rollback the transaction.

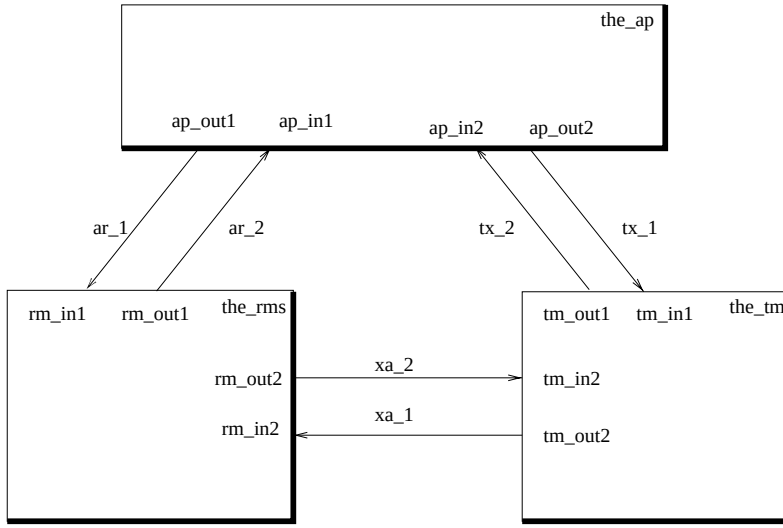

Fig. 3. Abstract SDTP architecture

4.2 SADL Description

SDTP has been described in SADL at an abstract level and then at successive levels down to one suitable for translation to a standard programming language. (In all there are 18 different descriptions of the architecture). The general architectural model is being used in the formal verification of the architecture.

Many kinds of formal analysis can be applied to the SDTP reference architecture. Our initial work just deals with the verification of the security enhancements to the basic DTP architecture. We assume that the underlying DTP protocol correctly implements the transaction, and our proof must show that the extensions enforce the "no read up, no write down" policy of multilevel secure systems.

The SADL text describing an abstract level of the SDTP reference architecture is given in Appendix A and figure 3 provides a diagram. This abstract level is similar to that used to illustrate proof-carrying architectures in [13], and we have mechanised the proof of multilevel security as outlined there.

4.3 Core Structural Semantics

The abstraction of the description in SADL into the core architectural model results in the following:

```
ap, rms, tm: COMP_TY
the_ap, the_rms, the_tm: COMP
direct: CONN_TY
ar_1, ar_2, tx_1, tx_2, xa_1, xa_2: CONN
ap_in1,ap_in2, rm_in1,rm_in2, tm_in1, tm_in2: IN_PORT
ap_out1,ap_out2, rm_out1,rm_out2, tm_out1, tm_out2: OUT_PORT
MK_COMP(ap,(:ap_in1,ap_in2:),(:ap_out1,ap_out2:) ) AND
MK_COMP(rms,(:rm_in1,rm_in2:),(:rm_out1,rm_out2:) ) AND
MK_COMP(tm,(:tm_in1,tm_in2:),(:tm_out1,tm_out2:) ) AND
INST(ap)(the_ap) AND
INST(rms)(the_rms) AND
INST(tm)(the_tm) AND
INST(direct)((:ar_1, ar_2, tx_1, tx_2, xa_1, xa_2:)) AND
CONNECTS11(ar_1,ap_out1,rm_in1) AND
CONNECTS11(ar_2,rm_out1,ap_in1) AND
CONNECTS11(tx_1,ap_out2,tm_in1) AND
CONNECTS11(tx_2,tm_out1,ap_in2) AND
CONNECTS11(xa_1,tm_out2,rm_in2) AND
CONNECTS11(xa_2,rm_out2,tm_in2)
```

The core semantics declares the types of the components and connectors, their instances, and connections. This describes the structural skeleton of the design. The semantic layer must describe the properties of concern and how these properties relate to the structural skeleton.

4.4 General Dataflow Semantic Layer

We are concerned with dataflow so we must have a definition of dataflow along with, if required, axioms defining how it can be analysed. The core definition states simply that data can flow from `c1` to `c2` if there is a connection between an output port of `c1` and output port of `c2`:

```
DirectFlow(c1,c2): bool =
  EXISTS conn, output, input:
  CONNECTS(conn,port(c1,output),port(c2,input));
```

If we now add an axiom stating that dataflow is a transitive property we can compute the resultant dataflows of the architecture using the transitive closure method introduced earlier. Due to the bi-directional connections between components, we compute that all intercomponent dataflows are possible including the fact that data can flow from the resource managers to the transaction manager and from the transaction manager to the application.

In the proof illustrating proof-carrying architecture of [13] an axiom is introduced stating that if secure data flows from the resource managers to the application then it flows via the connection labelled here `ar_2`. The apparent result of the dataflow analysis is that this condition is violated by the flow via the transaction manager. In fact, the semantic model of dataflow is insufficient; a more accurate model of secure dataflow is required. Our coarse definition of

dataflow does not distinguish between flow of control information, for example the return parameter of a procedure denoting success or failure, and the flow of arbitrary data. The connection between the transaction manager and the application manager only passes status and control information, and cannot carry secure data. (We are dealing with an abstract model; implementation issues such as presence of covert channels must be dealt with separately.)

4.5 Precise Dataflow Semantic Layer

A more accurate semantic analysis must be carried out based on the underlying structure and a semantics defining which connections can carry data. The basic definition of `Carries` states that a connection carries a certain kind of data between ports if it connects those ports and is able to carry such data:

```
Carries(conn,data,p1,p2): bool =
  CONNECTS(conn,p1,p2) AND
  CAN_CARRY(conn,data)
```

The flow of data between components via a connection and via an unnamed connection can then be defined:

```
CanFlowVia(conn,data,c1,c2): bool =
  EXISTS output, input:
  Carries(conn,data,port(c1,output),port(c2,input))
CanFlowDirect(data,c1,c2): bool =
  EXISTS conn, output, input:
  Carries(conn,data,port(c1,output),port(c2,input))
```

The resultant flow of data between components in the architecture can be computed using transitive closure of `CanFlowDirect`, or equivalently using a PVS inductive definition of `CanFlow`, stating the flow can be direct or indirect:

```
CanFlow(data,c1,c2): INDUCTIVE bool =
  CanFlowDirect(data,c1,c2) OR
  EXISTS c3:
    CanFlowDirect(data,c1,c3) AND
    CanFlow(data,c3,c2)
```

4.6 Architectural Analysis Result

It is straightforward to use these definitions to derive the theorem stating that if secure data flows from the resource managers to the application then it flows via the connection labelled here `ar_2`:

```
FORALL data:
 CanFlow(data,the_rms,the_ap) IMPLIES
  Carries(ar_2,data,port(the_rms,rm_out1),port(the_ap,ap_in1))
```

In the proof illustrating proof-carrying architecture of [13] the above result was introduced as an axiom. What we have shown here is that it can be derived as a theorem from the core structural model and a semantic layer describing precise dataflow.

4.7 MLS Semantic Layer

The final part of the formal analysis of this abstract description of the SDTP architecture is to add a semantic layer describing the properties of multilevel security (MLS). This will allow us to deduce that the architecture enforces the security policy. The basis of multilevel security is the labelling of data, components and ports with security levels. The following declarations and definitions are the essential parts of the MLS semantic layer:

```
LABEL: TYPE
;<=:[[LABEL,LABEL] -> bool]
Clearance: [COMP -> LABEL]
Clearance: [PORT -> LABEL]
label: [DATA -> LABEL]
SECURE_CHANNEL: [CONN -> bool]
SECURE_CHANNEL_AX: AXIOM
  FORALL (conn:(SECURE_CHANNEL)),data,p1,p2:
    Carries(conn,data,p1,p2) IMPLIES
     (label(data) <= Clearance(p2))
PORT_CLEARANCE_AX: AXIOM
  FORALL (comp:COMP),(input:(INPUT_OF(comp))):
  (Clearance(port(comp,input)) <= Clearance(comp))
LABEL_TRANSITIVITY_AX: AXIOM
  FORALL (l1:LABEL),(l2:LABEL),(l3:LABEL):
  ((l1 <= l2) AND (l2 <= l3)) IMPLIES
     (l1 <= l3)
```

The type of security labels is introduced along with an ordering on labels (by overloading `<=`). The security labelling of components, ports and data is given by `Clearance` and `label`. Secure channels are a subtype of connectors.

Three axioms are introduced, following the proof of [13]. The first states that a secure channel will only carry data to a port if the security label of the data is less than or equal to the clearance level of the port. The second states that the clearance level of input ports is less than or equal to the clearance level of the component. The final axiom states that the ordering of security labels is transitive.

Based on the above axioms and the semantic layer describing MLS one can deduce the following theorem:

```
Security_Policy: LEMMA
   FORALL data:
     Flows(data,the_rms,the_ap)  IMPLIES
      (label(data) <= Clearance(the_ap))
```

The result states that the architecture enforces the security policy: data can only flow from the resource managers to the application if the security label of the data is less than or equal to the clearance level of the application.

4.8 Summary

The architectural analysis of this abstract level of the SDTP architecture illustrates the general model and proof techniques. The design description is translated into the core structural model and semantic layers describing dataflow and security labelling. The analysis is architectural: it is based on the underlying structural model, augmented by the semantic layers.

5 Discussion

5.1 Related Work

The formal underpinnings of software architecture have been studied by many researchers. Topics addressed include theories of connection behaviour, formal models of composition, formal characterisation of non-functional properties. The semantics of the design interchange language ACME has been illustrated in [3]. Our work builds on and complements these efforts: our concern is to structure the semantics so as to provide a general tractable basis for mechanised proof. We expect that this lightweight layered approach can support many of the models and kinds of analysis developed by other researchers.

The ACME project is also based on the idea of a common basis for describing architectures with properties extending the core description. We follow the general design principles of ACME but are concerned with semantics rather than design information. The layers in our model are related to the use of properties in ACME; they differ in that the layers are semantic rather than based on syntactic design information. Thus, while an architecture incorporating different ADLs might result in different properties in ACME, they might all be mapped to a single semantic layer (describing dataflow, for example) in our model. The layered semantics fits well with the ACME approach and may provide a basis for complementing the design information with semantic information.

Various approaches are possible for the task of semantic integration of different ADLs. A very powerful and general approach consists of developing 'a semantic foundation for composition and interoperation of heterogeneous components' [9]. In this approach one might provide a means of representing the semantics of various ADLs in a common logical system, and also provide the mathematical infrastructure of metalogical or multi-model techniques for relating these heterogeneous semantics. Our approach is a complementary, pragmatic one, and provides a tractable basis to begin experiments in semantic integration without a large investment in theory and embedding of language semantics.

The layered semantic model means that the integration is achieved via a common structural model and via abstraction to common semantic layers. This

avoids full translation between ADLs; abstraction to common semantic layers provides the 'semantic interchange'. Other integration systems such as [1, 14] are much more comprehensive and tightly defined. Our approach is a lightweight but very general one which provides a basis for exploration of integration via the user defined semantic layers.

5.2 Further Work

This work explores a model and methods supporting formal analysis of software architectures. No new technology or new theory has been introduced, and the inherent difficulties of developing a description and abstractions of a complex software architecture remain. The work does suggest a useful layered semantic model, and argues for a lightweight approach to achieve effective architectural analysis. Further development driven by case studies, incorporating much automation, is needed as the basis of a useable tool.

The structured semantic model developed fits well with the open semantic framework of ACME and could be integrated with the ACME design interchange language and toolset. Some changes would be required. For example, in our model the point of interaction between a component and connector is identified as a single port entity while ACME supports a richer model of connectors where a connector interacts via roles and these roles are attached to the ports of components. The difference between semantic layers and properties has also been mentioned. Reconciling these and other differences is not inherently difficult but may involve trade-offs in the solution.

Our work has been motivated by the need for a general tractable semantic framework, and the main stimulus for further evaluation, refinement and improvement must be more case studies. Software architecture techniques offer benefits in dealing with large complex software systems. Formal analysis of such systems must have mechanised support and must be efficient, based on tractable models.

5.3 Conclusions

Our work has developed a general semantic framework for architectures that provides tractable models and supports architectural formal analysis. As well as supporting the SADL language, it may provide a formal model for other ADLs and support the semantic integration of heterogeneous designs using various ADLs. The work has grown out of a real case study and has been used in case study proofs. The approach is a pragmatic one, and seems to offer an effective basis for automated architectural analysis. Further case studies are needed to guide development of the model and automatation of the techniques.

Acknowledgements Our work benefits from interaction with many colleagues at SRI and beyond. The paper has benefited from incorporating useful suggestions made by the anonymous referees.

References

[1] Hubert Garavel. Open/caesar: An open software architecture for verification, simulation, and testing. In *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, March 1998.

[2] D. Garlan, R. T. Monroe, and D. Wile. Acme: An archiectural description interchange language. In *Proceedings of CASCON '97*, November 1997.

[3] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[4] David Garlan and Zhenyu Wang. A case study in software architecture interchange. Submitted for publication to the Workshop on Software and Performance 98, March 1998.

[5] John Herbert. Abstraction for architectural proof. SRI CSL Dependable System Archiecture Group, Working Paper, December 1998.

[6] V. Issarny and C. Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Proceedings of the Sixteenth IEEE International Conference on Distributed Computing Systems*, 1996.

[7] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[8] J. Magee, S. Eisenbach, and J. Kramer. *Modelling Darwin in the π-Calculus*, volume 938 of *LNCS*. Springer-Verlag, 1995.

[9] Jose Meseguer. Semantic foundations for compositions. DARPA ITO Project Summary, 1998.

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

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

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

[13] R. A. Riemenschneider. Checking the correctness of architectural transformation steps via proof-carrying architectures. In *Proceedings of SIGSOFT '98*, 1998. Available at `http://www.csl.sri.com/sadl/pca.ps.gz`.

[14] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. Technical Report ICS-TR-97-35, University of California, Irvine, Department of Information and Computer Science, aug 1997.

[15] M. Shaw, R. DeLine, D. V. Klein, T.L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[16] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC97*, pages 6–13, August 1997.

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

Appendix: Abstract Level of SDTP in SADL

```
x_open_abstract_df: ARCHITECTURE [ -> ]
IMPORTING ALL FROM Dataflow_style
BEGIN
  ar_requests: TYPE
  ar_resources: TYPE
  tx_commands, tx_responses: TYPE
  xa_commands, xa_responses: TYPE
COMPONENTS
  ap:  TYPE <= Function [ap_in1: ar_resources, ap_in2: tx_responses
                    -> ap_out1: ar_requests, ap_out2: tx_commands]
  rms: TYPE <= Function [rm_in1: ar_requests, rm_in2: xa_commands
                    -> rm_out1: ar_resources, rm_out2: xa_responses]
  tm:  TYPE <= Function [tm_in1: tx_commands, tm_in2: xa_responses
                    -> tm_out1: tx_responses, tm_out2: xa_commands]
  the_ap:  ap
  the_rms: rms
  the_tm:  tm
CONFIGURATION
  ar_1: CONNECTION =
          (EXISTS c: Channel<ar_requests>)
             Connects(c, the_ap.ap_out1, the_rms.rm_in1)
  ar_2: CONNECTION =
          (EXISTS c: Channel<r_resources>)
             Connects(c, the_rm.rm_out1, the_ap.ap_in1)
  tx_1: CONNECTION =
          (EXISTS c: Channel<tx_commands>)
             Connects(c, the_ap.ap_out2, the_tm.tm_in1)
  tx_2: CONNECTION =
          (EXISTS c: Channel<tx_responses>)
             Connects(c, the_tm.tm_out1, the_ap.ap_in2)
  xa_1: CONNECTION =
          (EXISTS c: Channel<xa_commands>)
             Connects(c, the_tm.tm_out2, the_rms.rm_in2)
  xa_2: CONNECTION =
          (EXISTS c: Channel<xa_responses>)
             Connects(c, the_rms.rm_out2, the_tm.tm_in2)
END x_open_abstract_df
```