CONTEXTUAL CONSTRAINTS – BREAKING THE "TO CONSTRAIN OR NOT TO CONSTRAIN" DILEMMA

R. Lakshminarayanan and Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science
Bangalore - India
lakshmin@irisa.fr & srikant@csa.iisc.ernet.in

Abstract

Software architectures are developed incrementally, and the development involves experimenting with a variety of components, interaction patterns and styles. Composition, substitution and refinement are three important operations that are performed during these design experiments. ADLs provide constructs to express composition, substitution and refinement. It is important to be able to specify that "X is composed with Y," but the fact "why X is composed with Y" is much more important and valuable, for they serve as the (only) guide line for the evolution of the system. Such design decisions are captured as constraints in the architecture specifications. Every design decision, specified as a constraint, is tightly bound to its context, and is often applicable only in that context. To be able to specify the context of a constraint at the finer granularity of operations, it should be possible to specify the operational context in which the constraint is applicable. In this paper we present contextual constraints, *as supported by our ADL TriSL, which permit tagging of constraints with operational contexts. TriSL blends the notion of type conformance with contextual constraints to ensure that the constraints are checked when (and only when) the corresponding design operations are performed.*

Keywords: *Software architecture description languages, Contextual constraints, composition, substitution, refinement, TriSL, software architecture, specification.*

1 Introduction

Software architecture development involves design and analysis of architectures. The design and analysis are inseparable as design decisions are validated and influenced by the results of the analysis. The development is done incrementally starting from an initial model annotated with properties. The model is modified and elaborated to incorporate the results of the analysis. This process is repeated iteratively till an architecture that meets the requirements of all the stake holders is reached.

In general, the architecture development process involves experimenting with a variety of component types, interaction protocols, and architectural styles. Examples of such experiments include determination of the usability of a component from a library, evaluation of a COTS (Commercial Off-The-Shelf) component for its suitability to the task at hand, determination of the correctness of the refinement of a particular component, and determination of when and how can two components be composed together. These design experiments can be considered as a set of operations performed on the architectural model. We note that composition, substitution and refinement are three important operations that are often performed during design experiments.

In this Section we introduce an example, which will be used to explain these operations. The goal of the example is to design an architecture for a system that will support remote print service. We start with an initial model of the architecture which consists two components, viz., `PrintServer` and `Client.` We identify the following characteristics of the `PrintServer`: it can accept request for printing documents; it can print only documents of the type PS and only at a speed of six pages per minute; it can be configured to handle a variety of printers. The `Client` component is characterized by its requirement to be able to print documents.

1.1 Composition

Composition, in the context of design, refers to the operation that combines the functionality provided by two components (or systems) to form a larger component (or a system). On analyzing the initial model for composition of `PrintServer` with `Client,` we observe that the composition operation requires us to determine when and

how can the Client and PrintServer be composed together? In this case, the answer to the question "when can they be composed together?" is straight forward: "they can be combined when the Client can send the documents for printing, in the format required by the Print-Server, viz., PS." However, the answer to the question "how can they be composed together?" requires us to identify the protocol used by PrintServer to interact with its clients and check whether Client can speak this protocol.

We decide to introduce a connector, RPS-Connector, that defines the protocol for interaction between the Client and PrintServer. Now, where do we place the constraint: "the Client can send only documents of type PS to the PrintServer," and when should this constraint be checked for? The answers are obvious but their implications are subtle. We can easily choose to place the constraint in the RPS-Connector, and then decide that the constraint should be checked *when* RPS-Connector is used to *compose* a Client and a PrintServer. The implication to be noted here is the context sensitiveness of the constraint. The constraint is designed for, and needs to be checked, when the operation performed is composition. For example, this constraint need not be satisfied when the RPS-Connector is refined, i.e. when the refinement operation is performed on the RPS-Connector. Other related questions are: Is the introduction of the new RPS-Connector permitted? How to constrain such additions of new architectural elements? When should such constraints be verified? Here, not only the constraint is dependent on the context, but it is also at a very finer granularity, viz., the addition of an architectural element.

1.2 Substitution

Substitution, in the context of design, refers to the operation that replaces an architectural element by a "functionally equivalent" element. Consider that we find two COTS components which claim to provide the functionality of print servers. Now we need to determine whether these components are substitutable for the PrintServer component. We can observe that, a component is substitutable for PrintServer only if the functionality provided by the component (described in terms of the document type it can accept, the number of pages it can print per minute, and configurability) is at least as much as that is guaranteed by PrintServer. Again we can see that the constraint is dependent on the operational context, substitution.

1.3 Refinement

Refinement, in the context of design, refers to the operation of elaborating the structure and functionality of an architectural element. The elaboration of an architectural element is called the *representation* of the element. Representations range from a whole system representing a single element, to a code module that implements an element. Now, consider that the PrintServer is refined into a system PS-Rep. PS-Rep is composed of a LoadBalancer component and a PrinterDriver component. Now, we need to determine whether this refinement is correct? Which in turn prompts to the question: When can one say that system PS-Rep is a correct refinement of Print-Server? In the case of our example, we can formulate a condition as follows. PS-Rep is a correct refinement of PrintServer only if the LoadBalancer component of PS-Rep can guarantee the speed requirement (six pages per minute) of the PrintServer and the PrinterDriver component of PS-Rep is configurable. The parameters of constraints on refinement will always be dependent on the representation. Hence, constraints on refinement are often applicable only in the context of the refinement operation and are not much relevant to the other contexts, for example, composition.

1.4 "To constrain or not to constrain"

On observing the context sensitive nature of the constraints that are encountered during architecture development, architects can be expected to document these constraints only when they have a mechanism to specify and automatically verify, such contextual constraints. Current mechanisms for specifying constraints do not permit contexts to be attached to constraints and the specification of constraints at the granularity of operations. Hence, here is a situation where architects have constraints that are meaningful only in a particular context and only at a particular granularity of operations. As architects often come across such situations and do not have a mechanism to specify contextual constraints they face the "to constrain or not to constrain" dilemma. And more often, they choose not to specify the constraint at all! As a result, a valuable information that can guide the evolution of the system is lost!

In the following sections we explain the contextual constraints feature of TriSL, and then explain the use of contextual constraints to constrain composition, substitution and refinement. Then we present the related work and conclude with pointers to future work.

2 Contextual Constraints

As shown in the previous section, constraints, that represent design decisions, are dependent on operational contexts. The operational contexts range from operations frequently performed in design experiments, like composition,

refinement etc. to internal decisions made by a specification analyzer, like a subtyping judgment. Architecture Description Languages (ADLs) are widely used for formal specifications of architectures, and they permit specification of constraints as a part of the architecture specification. Clearly, an ADL should facilitate the tagging of constraints with operational contexts, thereby allowing the architect to capture critical design decisions that would guide the evolution of the system. For capturing these design decisions, architects need a mechanism to specify and verify these constraints.

Broadly, the requirements for contextual constraints can be classified into those for specification and those for verification.

- **Specification:** It should be possible to specify constraints on any architectural element, like components, connectors, ports, roles, etc. and it should also be possible to specify the contexts in which these constraints are valid.

- **Verification:** It should be possible to automatically verify the satisfaction of the constraints required for any architectural element, in and only in the context in which the constraints are specified to be valid.

These requirements have significant ramifications on the design of the ADL and its support environment (type checker, analyzers, etc.). The requirements for specification require an ADL to explicitly identify the operational contexts, to which constraints can be associated. This in turn requires the ADL to be based on an operational model. The requirements for automatic verification of constraints require an ADL and its support environment to explicitly identify the operations which can be used as operational contexts in constraint specifications. The ADL support environment need to play an important role by exposing some of its operations (for example, type judgment) to be used as contexts in constraint specifications. These requirements can be satisfied by an ADL which treats architectures as artifacts that are *developed* incrementally and not as artifacts that are *specified*. This shift from the specification model to the incremental development model, will bring in the required notion of operations on the architectural structures and the notion of operational contexts to which constraints can be associated. Current ADLs are based on the specification model.

In the following section, we present our ADL, TriSL[1], which is based on an incremental development model, and defines operations on architectures as operations in the TriSL Abstract Machine, and supports specification and automated verification of contextual constraints.

[1]TriSL stands for **S**ystem **S**tructure **S**pecification **L**anguage.

3 Brief Overview of TriSL

```
PrintSystem :system extended with
   PrintServer :component extended with
      docType :string is "PS";
      speed :int is 6;
      configurable :bool is true;
      PrintService :port;
      DeviceAccess :port;
   end;
   PrintClient :component extended with
     PrintRequest :port;
   end;
   Printer :component extended with
     DataPort :port;
   end;
   RPS-Connector :connector extended with
    Client :role;
    Server :role;
   end;
   PrinterConnector :connector extended with
     Driver :role;
     Device :role;
   end;
   attach PrintServer.PrintService to
          RPS-Connector.Server;
   attach PrintClient.PrintRequest to
          RPS-Connector.Client;
   attach PrintServer.DeviceAccess to
          PrinterConnector.Driver;
   attach Printer.DataPort to
          PrinterConnector.Device;
end;
```

Figure 1. A Print System in TriSL

The basic building blocks of architectural descriptions are called *architectural elements* (or simply elements). For facilitating reuse of element descriptions ADLs support specification of *element types* (or simply types). **Component, port, connector, role,** and **properties** form the basic elements of TriSL. In addition to this, TriSL permits the specification of a **system,** a configuration of components and connectors. Figure 1 gives a TriSL specification of the PrintSystem example. The system consists three components, viz., `PrintServer, PrintClient` and `Printer,` and two connectors, viz., `RPS-Connector` and `PrinterConnector.` The `Printer` component and the `PrinterConnector` are new and the rest of the elements are same as in the example discussed in Section 1. The `Printer` component represents the hardware printer and the `PrinterConnector` represents the connection between the hardware and the `PrinterServer.` The configuration of these components and connectors, specified using a set of *attachments*, form the `PrintSystem.` For example, in Figure 1 the `PrintService` port of `PrintServer` plays the role of `Server` in the inter-

action defined by `RPS-Connector` and this is specified by "`attach PrintServer.PrintService to RPS-Connector.Server`."

In TriSL, every element has a type. For example, in Figure 1, the type of `PrintServer` is **component** and of `PrintService` is **port.** TriSL allows specification of element types using the keywords **compt, connt, portt,** and **rolet,** which respectively represent the kinds component, connector, port, and role types. In TriSL, architectural styles, which capture recurring patterns of system organization, are also considered as element types, and are defined using the keyword **style.** TriSL provides the following built-in property types: **int, bool, real,** and **string.** For ease of use, TriSL provides a set of empty types, viz., component, connector, port and role, which are of the kind **compt, connt, portt,** and **rolet** respectively. An element, which is an instance of a type, gets its structure and properties from the type it claims to be. Further, it can add more structure or properties by using the **extended with** construct. For example, the `PrintServer` component in Figure 1 claims to be of the (empty) type **component** and then adds all the structure (the ports) and the properties that characterize the component.

In TriSL, *types* form the basic mechanism for specification and verification of constraints. The constraints are specified as a part of the type specification and are validated in the context of the instance. The constraint language of TriSL is based on first order logic. An element $\mathcal{E}$ is judged of type $\mathcal{T}$ if and only if it possesses the structure required by $\mathcal{T}$ and it satisfies all the constraints specified by $\mathcal{T}$. The structure of a type are the elements and properties it defines. For example, for the connector type `RPS-ConnectorT` (figure 2) the role types `Client` and `Server` together with the property `protocol` constitute its structure. The constraints are tagged with the context in which they are valid. The TriSL type system, when making a type judgment, verifies only those constraints that are appropriate for the current operation. TriSL is based on structural type equivalence and structural subtyping. Hence, two types are equivalent if they have the same structure but different names. The name independence is up to the reordering of the fields of the type. A formal specification of the TriSL type system can be found in [2]. In the following sections we explain the specification and verification of contextual constraints with examples. As constraints are specified as a part of the type definitions, the examples are TriSL specifications of architectural element types.

4 Constraining Composition

Connectors mediate the interaction between components and serve as a medium for composition of functionalities. Hence, constraints on composition can be specified through connectors. Components are composed together through a connector, by *attaching* the appropriate ports of the components with the appropriate roles of the connector. Recall the composition constraints, formulated in Section 1.1, for the `PrintServer` example. The `Client` should be able to send the documents for printing in the "PS" format, and it should be able to speak the protocol understood by the `PrintServer.` We can specify these constraints by defining a connector type, `RPS-ConnectorT,` as given in Figure 2. As roles form the basis of attachments, it is appropriate to specify the constraints as a part of the role type definitions. The constraints in the role type `Client,` specify that any component which wants to play the role of a `Client,` should be able to send documents in the "PS" format, and should be able to speak the "Rprint-protocol". On the similar lines, the constraints in the role type `Server` specify that any component which wants to play the role of a `Server` should be able to accept documents of type "PS" and speak "Rprint-protocol".

Having specified the constraint, we need to define the context in which the constraint is valid. Composition is expressed as attachments between ports and roles. TriSL provides the **attach** operation, to attach a port to a role. Hence, the right operational context to validate the constraints on composition is the attach operation. Whenever an attach operation is performed on the roles (`Client` and `Server`) of an instance of `RPS-ConnectorT`, the constraints will be validated and only if the constraints are satisfied the attach operation will succeed.

```
RPS-ConnectorT : connt is

   protocol : string is "Rprint-Protocol";

   Client : rolet is
     onattach ensure
          ( Client.docType eq "PS" and
            Client.canSpeak eq "Rprint-Protocol" );
   end;

   Server : rolet is
     onattach ensure
          ( Server.docType eq "PS" and
            Server.canSpeak eq "Rprint-Protocol");
   end;

end;
```

Figure 2. Constraining composition

5 Constraining Substitution

The functionality of an element is specified in terms of the structure, properties and constraints on the structure and

properties. A component $\mathcal{A}$ can be substituted for $\mathcal{B}$ if they are functionally equivalent, i.e. if $\mathcal{A}$ possesses the same structure and properties as that of $\mathcal{B}$, and $\mathcal{A}$ satisfies all constraints defined for $\mathcal{B}$. The structure, properties and constraints of $\mathcal{A}$ and $\mathcal{B}$ can be abstracted and specified as types, say $\mathcal{A}_t$ and $\mathcal{B}_t$. Now the problem of substitutability can be reduced to checking whether $\mathcal{A}$ conforms to the type $\mathcal{B}_t$. In TriSL, type judgments include constraint satisfaction, hence any component that conforms to the type $\mathcal{B}_t$ can be substituted for $\mathcal{B}$. This can be generalized to: *any component that conforms to a type which is a subtype of $\mathcal{B}_t$ can be substituted for $\mathcal{B}$.* As the type system is based on structural subtyping, we can constrain substitution by constraining the subtyping. In fact, constraining substitution is just a direct use of constraints on subtyping. Constraints on subtyping can be used in conjunction with other contexts to form powerful constraints.

Figure 3 specifies a component type, `PrintServerT`, for the `PrintServer` component, discussed earlier. The constraints on the substitution of the `PrintServer` component (document type, speed and configurability) are specified as a part of the type definition. The constraints are specified to be valid on the context of a subtype judgment. Whenever the type system attempts to judge an instance to be of a type which is a subtype of `PrintServer`, the constraints will be validated in the context of the instance. The type of the instance is judged as a valid subtype of `PrintServerT` if and only if the constraints are satisfied. The identifier **self,** used in the constraint specification will be bound to the context of the instance.

```
PrintServerT :compt is

   docType :string is "PS";
   speed :int is 6;
   configurable :bool is true;

   PrintService : portt;
   DeviceAccess : portt;

   onsubtype ensure
       ( self.docType eq "PS" and
         self.speed eq 6 and
         self.configurable eq true );

end;
```

Figure 3. Constraining substitution

6 Constraining Refinement

Architectural elements are elaborated by refining the structure and properties. The elaboration is called the representation of the element. The main task is to refine the functionality by adding more detail. The representations range from a whole system to a code module that implements the element. Refinements specify a mapping from the element to its representation. This mapping forms the basis for verifying the correctness of the refinement. Figure 4 specifies the representation of the `PrintServer` component, discussed earlier. The representation consists two components, viz., `LoadBalancer` and `PrinterDriver` connected by a `DataPipe` connector. The `PrintService` port of `LoadBalancer` and the `DeviceAccess` port of `PrinterDriver` are bound to the respective ports of `PrintServer.` These bindings serve as the mapping from the `PrintServer` to its representation. Constraints on refinement use this mapping for relating the properties of the element and the representation. Note the property `speed,` of the `LoadBalancer` component is 4, whereas that of `PrintServer` component is 6. It is quite expected that load balancing might increase the processing time and hence decrease the speed. However, the acceptable speed range can be specified by the `PrintServer` and any representation of it, can be ensured to have a value for `speed` within the range.

```
PrintServer :component of PrintSystem is

   LoadBalancer :component extended with
        docType :string is "PS";
        speed :int is 4;
        PrintService :port;
        DriverAccess :port;
   end;

   PrinterDriver :component extended with
        Configurable :bool is true;
        DataAccess :port;
        DeviceAccess :port;
   end;

   DataPipe :connector extended with
       Producer :role;
       Consumer :role;
   end;

   attach LoadBalancer.DriverAccess to
          DataPipe.Producer;
   attach PrinterDriver.DataAccess to
          DataPipe.Consumer;
   bind LoadBalancer.PrintService to
        PrintServer.PrintService;
   bind PrinterDriver.DeviceAccess to
        PrintServer.DeviceAccess;
end;
```

Figure 4. Refinement of PrintServer

The properties `speed, docType,` and `configurability` define the important characteristics of

PrintServer and any representation should guarantee that it preserves these characteristics. Such characteristics can be specified as constraints in the definition of the PrintServer type, and can be specified to be valid in the context of refinement. As a part of refinement every representation provides a set of bindings. The **bind** operation provides the operational context for verifying the constraints on refinement. Figure 5 specifies a type for the component PrintServer. It specifies the constraints as a part of the port types PrintService and DeviceAccess. The **self** in the expression **self.comp,** refers to the port instance and **comp** refers to the component in which the port is contained. This notation allows the properties of a component to be accessed through its port. The constraint in PrintService ensures that the representation supports the docType "PS" and its speed is greater than 3 pages per minute. Further, the constraint in DeviceAccess ensures that the representation is configurable. The constraints are verified when a bind operation is performed on the instance of these ports, contained in an instance of the PrintServerT. The operation succeeds only when the constraints are satisfied.

```
PrintServerT :compt is

    docType :string is "PS";
    speed :int is 6;
    configurable :bool is true;

    PrintService :portt is
     onbind ensure
            ( self.comp.docType eq "PS"
              and self.comp.speed gt 3 );
    end;

    DeviceAccess :portt is
     onbind ensure
            ( self.comp.configurable is true );

end;
```

Figure 5. Constraining refinement

7 Other useful operational contexts

All the definitions (elements and types) and operations in TriSL are translated into a sequence of operations of the TriSL Abstract Machine (TAM). Figure 6 gives the operations defined in TAM. The structured operational semantics of the TriSL language in terms of these operations can be found in [2]. These operations form as a good basis for the operational contexts. Figure 7 lists the set of operational contexts supported by TriSL. There are operational contexts like **onconstruct** and **onsubtype** for which there is no corresponding operation. The choice of the set of operational contexts is motivated by the nature of the architecture design process, and is not solely based on operations.

OPERATION	MEANING
add	add an instance or type to another instance or type
del	delete an instance or type from another instance or type
attach	attach a port instance to a role (or vice versa)
detach	detach a port instance from a role (or vice versa)
bind	bind a port/role to its representation
unbind	unbind a port/role from its representation
set	set the value of an instance
fuse	fuse two instances or types

Figure 6. Operations in TAM

TAG	OPERATIONAL CONTEXT
onadd	add operation on mutable instances
ondel	del operation on mutable instances
onattach	attach operation on instances
ondetach	detach operation on instances
onbind	bind operation on instances
onunbind	unbind operation on instances
onconstruct	Declaration of instances
onfuse	fuse operation on instances
onsubtype	Implicit or Explicit subtyping
onall	All operations

Figure 7. The operational contexts supported by TriSL

The operational contexts **del, detach** and **unbind** can be used to specify constraints on the integrity of the structure of a system. The operational context **add** can be used to constrain the type of new elements that can be added to the existing architecture. For example, this context can be used to specify the constraint whether the RPS-Connector, discussed in the example in Section 1.1, can be introduced or not. Constraints that are valid on all contexts and need to be verified on all operations can be specified using the **onall** context. We do not claim that these operational contexts covers all possible contexts, but we do claim that they form a very useful set.

8 Related work

TriSL is designed to specify architectural structures, properties and constraints on them, we present here only those ADLs that have similar design goals.

Armani[5] supports design element types and property types. Two types of constraints are supported, viz., design invariants and heuristics. These constraints are specified as a part of the design element type specifications. Design invariants are constraints that need to be satisfied by all the instances of the element type. Heuristics are just design guidelines and are not required to be satisfied by all the instances of the element type. Armani provides a constraint language based on first order logic, to express constraints in type and style definitions. Armani does not permit association of contexts with constraints. The constraints are verified on all the operations performed on the instance.

SADL[6] is aimed at representing and reasoning about architectural hierarchies. SADL at the highest level, provides constructs to represent architectures, mappings and architectural styles. For modeling architectures, the following constructs are provided: component, connector, configuration, connections and constraints. Mappings are relations that define a syntactical interpretation from the language of an abstract architecture to the language of an concrete architecture. SADL permits two types of constraints to be defined as a part of a style definition, viz., constraints on well formedness of an architectural element and semantic constraints on refinement. SADL comes quite close to TriSL in terms of the expressibility of constraints, as it also supports constraints on refinement. However, it does not permit constraints to be tagged with the context in which they are valid. Further, the refinement constraints in SADL, can be specified in TriSL as a part of the type definitions and can be tagged with the operational context **onbind.**

C2 is an architectural style and has an associated ADL, C2-ADL[4], that provides constructs for specifying architectures that conform to the C2 style.While such style specific ADLs cannot be used to model other architectural styles, they do highlight the benefit of small, simple, formalisms highly tailored to a specific architectural style. The ArchShell tool, which is a part of the C2-ADL infrastructure, supports experimentation of dynamic architectures, by providing a set of operations to manipulate architectural structures and their runtime interconnection. From the perspective of providing a set of operations to manipulate architectural structures, C2-ADL together with the ArchShell is similar to the TAM. The operations provided ArchShell are for manipulation of the runtime structure of the architecture, whereas the operations provided by TAM are for manipulation of architectural structures at the specification time. C2-ADL is specific to a particular architectural style, and hence the constraints that are to be verified are predefined. C2-ADL does not support explicit tagging of constraints with operational contexts.

9 Conclusions and future work

Design decisions, represented as constraints, are closely tied to the contexts in which they are valid. If an architect does not have mechanisms to specify the context in which a constraint is valid, he is forced to rethink on whether to specify the constraint or not, and often they choose not to specify, than specifying a diluted version of the constraint, which is context independent. This results in loss of valuable decision decisions that can guide the evolution of the system. We have shown that the contextual constraints of TriSL, facilitates specification of operational contexts in which the constraints need to be valid. The incremental development model (architectural design treated as a set of operations, not as a specification) of TriSL, provides the appropriate contexts for the verification of these constraints. Contextual constraints, in general, are very expressive and permit specification and automatic verification of subtle constraints on the evolution of the system. So, with contextual constraints, the architects can break the "to constrain or not to constrain" dilemma, and can specify the valuable design decisions.

TriSL is implemented in C++, and has been successfully used to formalize the control architectures in automated manufacturing systems [3]. As of now, TriSL does not allow user defined operational contexts. We are working on providing user defined operations on architectural structures, and permit association of these operations as contexts for constraints evaluation. The incremental development model and the contextual evaluation of constraints are very useful for manipulation of architectural structures via tools. Tools like the ones used in [1], to find patterns in architectures can exploit the fact that only some constraints are valid in a particular context, and can reduce their search space.

References

[1] Kazman, R., and Burth, M. Assessing architectural complexity. In *Second Euromicro Working Conference on Software Maintenance and Reengineering*, 1998.

[2] Lakshminarayanan, R. TriSL: A Software Architecture Description Language and Environment. Master's thesis, Department of Computer Science and Automation, Indian Institute of Science, May 1999.

[3] Lakshminarayanan, R. and Srikant, Y.N. Formalizing control architectures in AMS. In *Second Nordic Workshop on Software Architecture*, August 1999.

[4] Medvidovic, N., Taylor, R.N., and Whitehead, E.J. Formal modelling of software architectures at multiple levels of abstraction. In *the proceedings of The California Software Symposium*, April 1996.

[5] Monroe, R. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University, School of Computer Science, 1998.

[6] Moriconi, M. and Riemenschneider, R. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, 1997.