Chapter 7

LINGUISTIC ISSUES

developed for
CSE327 - Advanced Software Engineering
Fall 1996

by

Jack Reisner, Cecilia Bastarrica
Imad Antonios, and Deb Smarkusky
Architectural Languages
Shaw & Garland, Chap. 7

- **Problem**: S/W Architectures described *informally*
  * Typically, informal diagrams used,
    supported by informal prose descriptions

- Moreover, informal descriptions used for
  * System *components*, and
  * Intercomponent *connections*
A Typical Architecture Diagram
(S & G, Ch. 7)

- Problems with this Figure?
  * Too ad hoc
  * Not standard

- Q: Are these concerns a reality?
  A: Consider some existing architectural descriptions...
Source:
These two examples add credibility to Shaw & Garlan’s claims, namely:

* Architectural descriptions are not standardized
* Architectural descriptions must rely heavily on informal documentation
• Architectural diagrams rely on *prose* descriptions

• At most, the *formal* documentation may include:
  * Type/parameter info
    * `C++ .h files or Ada package specs, e.g.
  * Module dependencies
    * `C++ includes or Ada with/use clauses, e.g.

• What’s missing? A way to differentiate between
  * More *critical* components
  * More *complex* components
  * Specialized component capabilities
  * ... etc.
VHDL: An analogy

- VHSIC Hardware Design Language (VHDL) is a computer language used to model very high speed integrated circuits

- Why was it developed? As circuit complexity grew, so did the need for a modelling tool which could
  * Model at various levels of abstraction
  * Be a standard language to support various tools

- VHDL Benefits
  * Very complex systems can be designed bottom-up or top-down
  * Standardization has been a catalyst for interoperable tool development

- S/W system complexity is also growing... could info engineers reap some of the same benefits?
VHDL Analogy, cont.

- Powerful tools can be designed when key domain building blocks are identified

- Consider the hardware realm:
  * Components: gates, flip-flops, adders, multipliers, shifters, ALUs, registers, etc.
  * Component connectors: bus, I/O pad, etc.

- What about software? Can S/W domain be so neatly categorized?
Shaw & Garland specify 5 common components

- **Computational**: Has no retained state (e.g., math functions, filters, transforms, etc.)

- **Memory**: Is a shared collection of persistent structured data (e.g., databases, symbol tables, etc.)

- **Controller**: Governs timed sequences (e.g., schedulers, synchronizers, etc.)

- **Link**: Transmits info between entities (e.g., user interfaces, comm links, etc.)

- **Managerial**: Has a state and set of operations (e.g., abstract data types, servers, etc.)
Shaw & Garland specify 6 common component connectors

- **Procedure Call**: Single control thread (e.g., standard procedure and function calls)

- **Dataflow**: Independent processes interact through data streams (e.g., UNIX pipes)

- **Implicit Invocation**: Computations invoked by non-explicit event occurrences (e.g., event systems)

- **Message Passing**: Interaction via explicit data handoffs, (synchronous or asynch.) (e.g., TCP/IP)

- **Shared data**: Concurrent manipulation of data by components on the same data space (e.g., blackboard system, multiuser databases)

- **Instantiation**: Instantiator uses capabilities of instantiated definition (e.g., use of abstract data types)
Architectural Languages: Issues

- Is there enough commonality to develop an architectural language?
  - Current design methodologies are too informal
  - Programming languages are too low-level

- If an architectural language was developed, what would be the developer’s goals?
Architecture-Description Languages

System to support architectural design:

- models, notations and tools
- large-scale, high level designs
- adaptation of designs to specific implementations
- user-defined or application-specific abstractions
- principled selection of architectural paradigms

Close interplay between language and environment:

- language necessary for precise descriptions
- environment for usable and reusable descriptions
(1) Composition

*It should be possible to describe a system as a composition of independent components and connections.*

Combining independent elements into larger systems with explicit, abstract composition rules:

- the language must allow to modularize and assemble a system into and from its independent elements
- separation of implementation choises from architectural structure
- entities of an architectural description are primitives at one level and composite structures at a lower level
(2) Abstraction

*It should be possible to describe the components and their interactions in a way that clearly and explicitly prescribes their abstract role.*

- abstraction - description of design elements in a level that matches the intuitions of designers.
- abstraction supress unnecessary detailed yet revealing important properties
- architectural design requires abstraction to assign roles to each element in the structure: kind of elements and relationships.
(3) Reusability

*It should be possible to reuse architectural patterns in different architectural descriptions.*

- many languages allow reuse but very few allow generic patterns of components and connectors

- family of system architectures: collection of architectural elements along with certain constraints on the semantics

- architectural components need further instantiation

- the need of reusability goes considerably beyond the capabilities of modular languages
(4) Configuration
Architectural descriptions should localize the description of system structure, independently of the elements being structured. They should also support dynamic reconfiguration.

- configuration permits us understand and change elements of an architectural description without lookin into their implementation

- dynamic reconfiguration is to change the structure at runtime

(5) Heterogeneity
It should be possible to combine multiple, heterogeneous architectural descriptions.

- combine different architectural patterns in a system

- desirability of combining component written in different languages
(6) Analysis

*It should be possible to perform analyses of architectural descriptions.*

- support for automated and non-automated reasoning about architectural descriptions

- special purpose analysis tools and proof techniques for each feature of architecture

- existing connection languages allow only type checking analyses

- enhanced analyses are important for dynamic properties

- no single semantic framework will suffice for architectural analysis
Problems with Existing Languages

(1) Informal Diagrams

- generally box and line diagrams (boxes for components, lines for connections)

- the meaning of boxes and lines vary, even within one diagram

- different shapes represent different kinds of components, different connections can almost never be represented.

⇒ high abstraction level, captures intuition, cannot be analyzed, not a clear relation with implementation, and rarely reused

(2) Modularization Within Programming Languages

- Simula classes, CLU clusters, Alphard forms, Ada packages

- module interface declares exports and imports

- modularization is successful for programming in the large:
  - reduces complexity through abstraction
  - permits cooperative work
  - incremental compilation
(a) Composition

- hierarchical decomposition, but poor support for independent composition of structural elements
- intermodule connection by name matching
- imports and exports definition in module interfaces forces connections to be part of module definition: poor reuse
- imports and exports definition confuses algorithms with connections

(b) Abstraction

- system structure has to be expressed using language primitives
- primitives for interconnections are usually few
- most form of interconnections can be built in terms of the programming language, but:
  - makes designers think in terms of the language primitives
  - reduces reusability
  - limits the level of abstraction of connections
- connections are not explicit at the architectural level
(c) Reuse

- library modules are used importing them explicitly without an alternative of implementation, rather than just their capabilities

- no reuse of patterns of composition

(d) Configuration

- connections defined all along the system module definitions

- difficult or impossible to understand or analyze the whole structure

- make files: show dependencies but not their nature
(e) **Heterogeneity**

- difficult to combine modules written in different languages

- even more difficult to use connections that are not primitives of the languages

(f) **Analysis**

- analysis are not supported mainly because it is not easy to see what the actual structure of the system is

- name matching is not enough to check consistency of connections
(3) *Module Interconnection Languages*

- alternative to define interconnections in the module interfaces
- separate definition of structure and components
- they share most of the drawbacks of programming languages

(4) *Languages for Alternative Forms of Interaction*

- some notations have been designed to support more complex interactions (UNIX pipes, event broadcast, Ada rendez-vous)
- only a limited set that just extends the vocabulary

(5) *Notations for Specialized Architectural Styles*

- special systems have been designed to be able to specify certain kinds of architectures
- the major drawback is that they do not support heterogeneity

(6) *Emerging Architectural Description Languages*

- some (new) languages are developed to achieve all the goals here mentioned
Software System Composition

Components:

- Components encompass the computation and the state of the system
- A component has an interface specification that defines the functionality of the resources it defines
- The visible entities of a component constitute its interface points

- **Primitive vs. Composite:**
  - *Primitive components* directly translate to programming units
  - *Composite components* can be further refined at the design level
Connectors

Connectors mediate the interaction of two or more components

A connector has a protocol specification defining its properties

**Protocol specification components:**

- Rules about types of interfaces to interact with
- Specification for interaction properties
- Rules about order of events
- Commitments about interaction (e.g. Performance)

Examples of primitive components:

- Remote procedure calls (RPC)
- Network Protocols

Implementations of connectors as:

- Procedure calls associated by a linker
- System functions of operating systems
- Library code
Separation of Components & Connectors

*Why separate components and connectors?*

1. Connectors may require complex specifications and definitions
2. Connector definition should be localized:
   - Support for analysis during design
   - Support for maintenance
3. Connectors can be abstract:
   - Support for parameterization
4. Connectors as independent entities:
   - Connectors may mediate relations among dynamically changing components
5. Components as independent entities:
   - Support for encapsulation
   - Interaction through interface points
6. Reuse of patterns of composition:
   - Definition of generic patterns (e.g. client-server, filter)
Motivating an Architectural Language

- Component and connector concepts are fundamentally different
- Poor support for intra-component connection semantics by programming languages
- Lack of current modeling languages

Potential language capabilities:

- Definition of connector semantics and composition
- Support for independence of entities
- Support for rules with asymmetry, multiplicity, locality, abstraction, and naming
- Definition of type structures for components and connectors
- Support for analysis of properties (e.g. security, performance)
- Specification of rules for architectural abstraction
**Language Structure**

- Need for *separate* but parallel constructs for components and connectors
- Language constructs are typed

<table>
<thead>
<tr>
<th>Element</th>
<th>Component</th>
<th>Connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>Interface</td>
<td>Protocol</td>
</tr>
<tr>
<td>Type</td>
<td>Comp. type</td>
<td>Conn. type</td>
</tr>
<tr>
<td>Unit of Assoc.</td>
<td>Player</td>
<td>Role</td>
</tr>
<tr>
<td>Implementation</td>
<td>Implem.</td>
<td>Implem.</td>
</tr>
</tbody>
</table>

Figure. Structure of an Architectural Language

- Implementation of non-primitive elements consists of parts list, composition instructions, and related specifications
- Name matching no longer needed
- Need for openness of specifications to construction and analysis tools
Connector Semantics (1)

Connectors as first-class entities:
- Encapsulate implementation details
- Same-type connectors may share same code and/or data

Requirements for connector specification:
- Guaranteed delivery of packets
- Event ordering restrictions
- Incremental production and consumption rules (pipelines)
- Distinction between roles of clients and servers
- Parameter matching for conventional procedure calls
- Restrictions on parameter types in RPC

Connector specification may be achieved through formal notation such as Hoares' CSP formalism (sec. 8.3)
Connector Semantics (2)

A connector may define:
- *Asymmetrical roles* - e.g. single owner of data for multiple users
- *Symmetrical roles* - defined by the ability of all system entities to recognize and generate events

**Cardinality of connectors:**

- *Binary* - e.g. user and definer as two roles of a protocol
- *N-ary* - *Symmetrical* - e.g. Multicast
  Asymmetrical - e.g. Client-Server

**Implementation issues:**

- Connectors often implemented as a set of procedures
- Rules and assumptions about how procedures are used are implicit

Need to have **explicit** restrictions:
- Expresses execution order
- Decouples selection of ADT from selection of implementation
Architectural Type Structures

- Architectural types describe the expected capabilities of components and connectors
- Should contain enforcement power to potentially restrict ways to use the constructs
- Type checking is needed when associating roles of connectors to interface points of components

Type checking issues:

What *could* happen when associations don’t quite match?

- Associate anyhow
- Rearrange or reformat information
- Convert data to/from shared from (standard interchange format)
- Convert data of one component to the form expected by another (pairwise compatibility)
- Insert conversion module
- Just say NO!
Explicit Architectural Notations

- **Reflections:**
  - Construction of composable systems is fundamentally different than programming
  - One goal is to build a system that has an enduring existence in some larger environment

Why *explicit*?

- Localization of information about interactions
- Introduction of abstractions for interactions
- Separation of architectural concerns from programming concerns
- Enhancement of reuse opportunities through "mild" type checking (at design level)
- Clarification of conditions where programming languages can be mixed
- Prospects for improved analysis of design and support of large software systems
Implicit Invocation

Concept

- Reactive integration and selective broadcast
- Extends explicit invocation
- Modules permitted to declare and announce events
- Register modules to receive announced events by associating one of its procedures with each event of interest
- Other modules register interest in event by associating procedure with the event
- When event is announced, system is responsible for calling the procedures registered for the event

Advantages

- *Reuse* - Integrate modules by registering interest in event
- *Evolution* - Replace module without affecting the interfaces of modules that implicitly depend on it
Event Specification Language Example

for Package_1
    declare Event_1
        X: Integer;
        Y: Package_N.My_Type;
    declare Event_2

    when Event_3 => Method_1 B
end for Package_1

for Package_2
    declare Event_3
        A, B: Integer;

    when Event_2 => Method_4
    when Event_1 => Method_2 X
end for Package_2

for Package_3
    when Event_2 => Method_3
    when Event_1 => Method_4 Y
end for Package_3

Clause specifications:
"for"   - identify package for event
"declare" - specify event that package will announce
"when"   - which procedures are to be invoked, and the parameters that are to be passed, when event is announced
Generated Spec for Event_Manager

with Package_N;
package Event_Manager is
  type Event is (Event_1, Event_2, Event_3);

  type Argument (The_Event : Event) is
    record
      case The_Event is
        when Event_1 =>
          Event_1_X : Integer;
          Event_1_Y : Package_N.My_Type;
        when Event_2 =>
          null;
        when Event_3 =>
          Event_3_A : Integer;
          Event_3_B : Integer;
        when others =>
          null;
      end case;
    end record;

  procedure Announce_Event(The_Data: Argument);
end Event_Manager;
Generated Body for Event_Manager

with Package_1;
with Package_2;
with Package_3;
package body Event_Manager is

  procedure Announce_Event(The_Data: Argument) is

    begin
      case The_Data.The_Event is
        when Event_1 =>
          Package_2.Method_2(The_Data.Event_1_X);
          Package_3.Method_4(The_Data.Event_1_Y);
        when Event_2 =>
          Package_2.Method_4;
          Package_3.Method_3;
        when Event_3 =>
          Package_1.Method_1(The_Data.Event_3_B);
        when others =>
          null;
      end case;
    end Announce_Event;
end Event_Manager;

Event Announcement -

Announce_Event(Argument'(Event_1, X_Arg, Y_Arg));
Event Definition

How are events defined?

1. **Fixed event vocabulary**
   - User is not allowed to declare new events

2. **Static event declaration**
   - Events introduced and fixed at compile time

3. **Dynamic event declaration**
   - New events declared dynamically at run time

4. **No event declarations**
   - Components can announce arbitrary events

√ Why **static event declaration**?
   - Efficient implementation as Ada enumerated type
   - Compile-time checking of declaration and uses
   - Predictability through static checking

Where are events declared?

1. **Central declaration** - Events declared at a central point and used throughout the system

2. **Distributed declaration** - Events declared by each module and each module declares the events it expects to announce
Event Structure

How are events structured?

1. **Simple Names**
   - Simple names without parameter information
   - Systems that use events as interrupts

2. **Fixed parameter lists**
   - All events have same names and parameters
   - Combined with set of system-defined events

3. **Parameters by event type**
   - Each event has fixed list of parameters
   - Type and number of parameters may differ by event

4. **Parameters by announcement**
   - Specify parameter list when announced
   - Receiver must decode parameters at run-time
   - May lead to unpredictable and undisciplined systems

✓ Why **parameters by event type**?
   - Easy to use in system construction
   - Easy to understand component interaction
   - Variant records represent parameters (Ada)
Event Bindings

When are events bound to procedures?

1. **Static event bindings**
   - Events bound to procedures at compile time

2. **Dynamic event bindings**
   - Event bindings created dynamically
   - Components *register* for events when they wish to receive them
   - Components *deregister* when they are no longer interested

√ Why **static event binding**?
   - Dynamic binding decreases predictability of system
   - Dynamic binding introduces race conditions at run time
   - Ada83 provided no convenient means for pointer to subprograms
   - Ada95 allows procedures as parameters
Event Bindings - continued

How are event parameters passed to the procedures?

1. **All parameters**
   - Invocation passes all specified parameters (number, type and order)
   - Requires conspiracy between designers of event and invoked procedure

2. **Selectable parameters**
   - Implementor specifies which event parameters are passed and in what order

3. **Parameter expressions**
   - Invocation passes results of expressions computed over the parameters of the announced event

✓ Why **selectable parameters**?
   - Provided balance between flexibility and ease of implementation
   - Reusability because of freedom in matching events to procedures
   - Argument list built directly from the event-binding declaration
Event Announcement

How are events announced?

1. **Single announcement procedure**
   - Single procedure to announce any event, which accepts a variant record with event type and arguments

2. **Multiple announcement procedures**
   - One announcement procedure per event name
   - Procedure accepts exactly the same parameters as the event

3. **Language extension**
   - Provide an *announce* statement as a new kind of primitive to Ada
   - Use preprocessor to conceal actual Ada implementation

4. **Implicit announcement**
   - Events to be announced as a side effect of calling a given procedure
   - Used as triggering mechanism for databases
   - Events announced without changing the module that caused announcement to happen

Why **single announcement approach**?
- All event announcements look similar
- Wanted to stay as close to pure Ada as possible
Concurrency

What is a component?

1. **Package**
   - A component is a package
   - An invocation is a call to a procedure in the package interface

2. **Packaged Task**
   - A component is a task (with an interface in a package specification)
   - An invocation is a call on an entry in the task interface
   - Permits concurrent handling of events

3. **Free Task**
   - A component is a task, built inside Event_Manager package
   - An invocation is a call on an entry in the task interface
   - Permits concurrent handling of events

✓ Why **package**?
   - Avoid synchronizing task of Event_Manager
   - Easier to develop single thread of control systems
Delivery Policy

When an event is announced, which procedures are invoked?

1. **Full delivery**
   - Announced event causes invocation of all procedures bound to it

2. **Single delivery**
   - An event is handled by only one of a set of event handlers
   - Provides a form of "indirect invocation" rather than "implicit invocation"

3. **Parameter-based selection**
   - Uses the event announcement’s parameters to decide whether a specific invocation should be performed

4. **State-based policy**
   - Associate a "policy" with each event binding
   - Given an event of interest, policy determines effect
   - Policy can ignore event, generate new event, or call a procedure

√ Why **full delivery model**?
   - Most straightforward analysis
   - Add policies in the future to provide flexibility
Evaluation

Background
- Masters-level software engineering course
- Average of 5 years experience
- Most familiar with Ada

Advantages
- Nature of events fit well with abstract model
- No conceptual problems in transferring abstract understanding of implicit invocation to use of implementation

Disadvantages
- Problems with debugging preprocessed code
- Lack of support for concurrency in design
- Absence of dynamic event declaration and binding

Summary
- Shown by example how to add implicit invocation to a statically typed, module-oriented programming language
- Elaborated design space for this approach and showed how decisions are affected by the constraints of the programming language
Reading List

Articles


*Garlan, David; Shaw, Mary; Okasaki, Chris; Scott, Curtis; Swonger, Roy.* “Experience with a course on architectures for software systems”. Proceedings of the Sixth SEI Conference on Software Engineering Education, pp. 23-43. Springer Verlag, LNCS 376, October 1992.
Articles - continued

Garlan, David; Perry, Dewayne E.. “Introduction to the Special Issue on Software Architecture”. IEEE Transactions on Software Engineering Vol. 21, No 4 269-274, April’95.


Books


Perry, Donald. VHDL
