CONNECTING THE DOTS: NATUALLY REPRESENTING YOUR SCA CONNECTIONS

John Hogg (Zeligsoft, Gatineau, QC, Canada; hogg@zeligsoft.com)
Francis Bordeleau (Zeligsoft, Gatineau, QC, Canada; francis@zeligsoft.com)

ABSTRACT

The Software Communication Architecture (SCA) standard and associated documents have no standard graphical representation of most types of Domain Profile connections. Connections within applications are well represented (at least at an informal level). However, connections to services or between application components and logical devices in underlying platforms are not. This causes errors not only in communication between humans, but also in delivered XML descriptor files. There is a simple, natural, graphical notation for depicting these connections. Developers who write or generate XML from these diagrams can deliver better quality profiles faster.

1. INTRODUCTION

The SCA standard itself [1] has no graphical notation for defining connections in profiles, or sets of XML descriptor files. Only the XML syntax and semantics are normative. Unfortunately, XML is hard to read, hard to write and simply not a good way for humans to express their intent.

This is recognized in the SCA Developer's Guide[2], which shows connections graphically in a notation borrowed from UML 2.0. Developers see boxes representing components, ports through which the components communicate on their boundaries, and lines representing connections between ports. A few boxes and lines crisply define what hundreds of lines of XML will obscure to normal minds.

The SCA Developer's Guide does an excellent job as far as it goes. Unfortunately the SCA connection vocabulary is very rich, and the Guide's graphical notation only handles a small part of this: the static connections between component instances within a single application. Connections between application components and logical devices or services are poorly represented today.

There is a simple solution to this problem based on the concept of "freestanding ports". A few simple concepts and icons express the entire vocabulary of SCA connections. The surface simplicity comes from solid formal underpinnings in the UML notion of role modeling, and the notation can be represented as a UML profile. This paper presents that solution.

2. BACKGROUND

Most software-defined radio (SDR) experts will be familiar with the basic concepts of the SCA but they will be briefly described here as a basis for the rest of this discussion. SCA architectures are based on a few simple concepts: components (including devices), interfaces, ports and connections.

Applications (or waveforms in the SCA context) are the complete units of software functionality that turn a hardware platform into a radio. They are constructed from communicating software *components*.

The component is the fundamental SDR software building block. An SCA component is a modular unit of software that encapsulates its contents behind interfaces and ports.

Developers deliver components as code (compiled into binary form) and XML descriptors or metadata describing components. The component is a reusable artifact and the component instance is its use.

Components are used in many environments. They are a first-class element in the UML (Unified Modeling Language)[3]. The representation of components in the SCA Developer's Guide is influenced by the UML in the way that it shows ports on components.

A *port* is a *conjugated* interface. A port can specify an interface that the component provides to its environment or it can specify an interface that the component requires from its environment. The direction of a port is its *conjugation*: a component provides an interface to its environment through a *Provides* port and uses interfaces provided elsewhere in its environment through a *Uses* port. An *Interface* on a component can be viewed as another type of conjugation for port matching purposes: a connection must have a Uses port at one end and a Provides port or Interface at the other.

The Provides conjugation is intuitive to software developers who haven't used ports. The Uses conjugation may be less so. The component does not communicate directly with its environment: it communicates with its ports. The Uses port is an outbound window that provides an interface. The component that responds to that interface is not directly visible to the component.

Ports are also *directed* interfaces. Internally, component may provide the same set of operations through two ports, but may handle an operation request in different ways depending on the port. For instance, a TankController will handle a `transitionHigh`() operations from sensors at the top and bottom of a water tank in different ways.

How does the controller in the example receive messages from the sensors? The architect wires together two ports with compatible interfaces and conjugations by a *connection*. (A Uses port can be connected to a Provides port or an Interface.) Ports only communicate through connections; a port cannot communicate to another port or interface unless they are connected. In this way the TankController can know who called the `transitionHigh()` operation.

An application does not run in a vacuum. A set of applications execute on a hardware *platform* or set of devices: general-purpose processors (GPPs), field-programmable gate arrays (FPGAs) and digital signal processors (DSPs). The applications are designed to be reusable across a variety of platforms, so application components do not directly drive device hardware. Instead, the devices are controlled through software *logical devices* that are platform-dependent components.

3. THE PROBLEM

If all connections were made between two clearly-defined ports on two known components they would be easy to specify and understand. However, the components at either end aren't always known or knowable to the application architect.

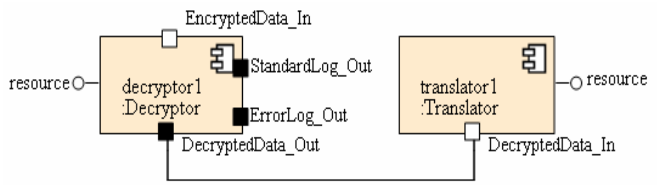

Figure 1

A component in an application can make a connection to a service or a logical device on a platform. Services (such as logs or file managers) are typically implemented by a core services team, not application developers. While the delivered application may depend upon them, it does not contain detailed information about how or where they are implemented.

Similarly, applications are intended to be portable across hardware platforms yet must make use of hardware logical devices. The SCA therefore has constructs for defining connections to devices that do not appear in any application descriptor file.

The SCA connection specification problem is complex. Freestanding ports address this problem.

4. REPRESENTING SCA CONNECTIONS WITH FREESTANDING PORTS

This section introduces the concept of a "freestanding port" and describes graphical representations for each type of SCA connection.

As explained above, for many connection types the component participating in the connection is not part of the software profile or not statically determinable. The natural way to depict the connection is therefore to separate the endpoint from its component.

This gives rise to the concept of a freestanding port: a port that is not associated graphically with any component in an application assembly or any logical device in a node definition. A freestanding port can represent access to a service such as a log. Alternatively, it can represent a port on a logical device of the underlying platform.

Since the component instance owning the port is not known at connection definition time, it is not represented. The idea of having a port divorced from any component may concern UML modeling and metamodeling experts. UML 2.0 ports are very much part of a "class with structure" and there is no way in which a UML port can exist on its own. If you are such an expert, relax. There is a clear and simple mapping from freestanding ports to UML standard metamodel definitions and the missing components are really there after all. Freestanding port compliance with the UML is described later.

We'll now see how freestanding ports allow the clean definition of all types of SCA connection. For completeness, we'll start with the simplest connections based on explicit components and build up from there.

SCA connection ends are essentially independent of each other (given appropriate conjugation). We will therefore concentrate on one end of each connection that we review. The first and simplest of these is a connection between ports on two component instance references.

Component Instance Reference Connections

Component instance reference connections are the standard way of specifying an intra-application communication path. The idea is simple: two components each have (compatible) ports and the application joins them with a connector. Examples of this type of connection appear in the SCA Developer's Guide. Figure 1 shows such a connection. Equivalent connections can be made between logical devices in a platform. For simplicity, only application connections will be discussed here.

Using connections between component instance reference ports makes the architecture of an application explicit. This is normally a best practice. Explicit architectures help all team members understand what is being implemented. Additionally, the validity of the connection can be statically validated. The component instances and their ports appear in the software or platform descriptor files.

The disadvantage of using component instance references is the lack of flexibility in architectural connections. The

```
<!--Connection between [Component decryptor1, Port DecryptedData_Out] and [Component
translator1, Port DecryptedData_In]-->
<connectinterface id="DCE:0ec07bec-6f9d-4c4f-b517-b9289951ad69">
     <usesport>
         <usesidentifier>DecryptedData_Out</usesidentifier>
         <componentinstantiationref refid="DCE:80286241-45f9-4296-ba4d-7ecbaa6f8549" />
     </usesport>
     <providesport>
         <providesidentifier>DecryptedData_In</providesidentifier>
         <componentinstantiationref refid="DCE:8d7f785a-b0c9-4e88-9c0a-3178395a3123" />
     </providesport>
</connectinterface>
```

Figure 2

architect must specify at design time who is connected to whom. Where this may cause problems, mechanisms such as event channels (described below) should be used. However, component instance reference connections should be the first type of connection considered when designing an application.

Component instance reference connections are easy to understand when they are displayed in the graphical view of an application or node. They are much harder to understand in XML descriptor files. The XML equivalent to Figure 1 is shown in Figure 2. The refid values indicating the component instances are both unambiguous and unintelligible to a human. However, this is not a problem when XML is generated from a tool instead of being written by hand.

Component instance reference connections are a kind of "base case" for SCA connectivity. *All the other connection types are better represented using freestanding ports*.

For consistency and simplicity, a component instance reference will be used at one end of each of the following connection descriptions. However, freestanding ports may be connected to each other.

For reasons of space, findby namingservice connections (and their limited utility) are not described in this paper. They are fully explained in [4]. They are also naturally represented using freestanding ports.

Services

Every architectural framework includes a set of services that its components may use. Some of these are tightly integrated into the framework itself; others may be plugged in to customize the framework for its environment. The SCA defines a set of standard service types but allows some actual services to be selected by name. The standard service types are filemanager, log, namingservice and eventchannel.

The eventchannel service is treated separately in the next section. Access to the other services in the SCA is similar in general (although there are some minor differences in the details of name semantics) and they can be considered together. The log service is a natural example.

Figure 3 shows two connections from a component

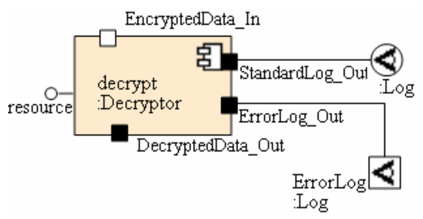

Figure 3

instance's ports to log services. In the top part of the diagram the standard log service is implemented as an interface. The "eye" icon suggests the "lookup" nature of the connection.

The top part of Figure 3 represents the XML shown in Figure 4. Services are specified using the domainfinder findby construct. In this example no name is specified so the interface can connect to any available log.

Services can also be implemented as ports as shown in the lower part of Figure 3. The port conjugation is Provides. The XML for this connection is shown in Figure 5.

In this case the connection is made to a service identifying itself as ErrorLog. A framework can be extended in this way to provide multiple log (or other) services to address various needs.

An SCA service is not part of an application and does not appear directly in the application architecture or in the XML profile describing it. The service belongs to an architectural layer defined at the platform level. The freestanding port is an access point through the layer to the service provided by

```
<!--Connection between [Component decrypt, Port StandardLog_Out] and [Free-Standing
             Interface]-->
<connectinterface id="DCE:56fab9f9-5a24-4379-84bb-d3631629414c">
    <usesport>
        <usesidentifier>StandardLog_Out</usesidentifier>
        <componentinstantiationref refid="DCE:7f081cab-80ff-48a3-9e46-d01f8c3a74b9" />
    </usesport>
    <findby>
        <domainfinder type="log" />
    </findby>
</connectinterface>
```

Figure 4

```
<!--Connection between [Component decrypt, Port ErrorLog_Out] and [Free-Standing Port
       ErrorLog]-->
<connectinterface id="DCE:c7185ca3-f1a0-4bb3-9321-a7ae5fd07d4a">
    <usesport>
        <usesidentifier>ErrorLog_Out</usesidentifier>
        <componentinstantiationref refid="DCE:7f081cab-80ff-48a3-9e46-d01f8c3a74b9" />
    </usesport>
    <providesport>
        <providesidentifier>ErrorLog</providesidentifier>
        <findby>
            <domainfinder type="namingservice" name="ErrorLog" />
        </findby>
    </providesport>
</connectinterface>
```

Figure 5

the layer underneath.

Event Channels

An event channel provides architects with completely flexible communication architectures. Event channel clients send or receive messages, but they don't know where the messages are going to or where they're coming from. Publish/subscribe is another name for this pattern. It supports the connection of potentially many channel content providers to many channel content consumers.

Event channels are a building block for many software designs. For instance, they allow a component to report a change in its state to its dependents without having to know who those dependents are. In one concrete example, a component can report low signal strength to an event channel. One consumer of the channel can use this information to search for another signal source while another consumer can initiate an antenna retuning. The source of the event does not need to know that there are multiple possible signal sources or a tunable antenna,

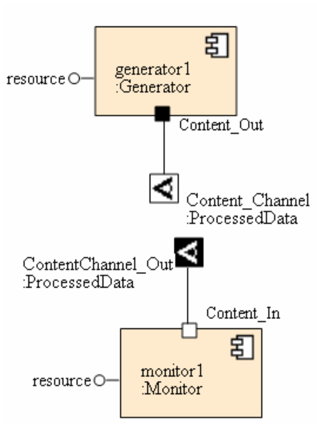

Figure 6

simplifying its design and portability.

Event channels are SCA services like naming service, log and file manager. Therefore, event channel graphical and XML representations closely resemble the other services. However, event channel conjugations differ from those of the other services. Instead of being Interfaces or Provides ports, event channels are represented as Provides ports to which channel content suppliers write messages and Uses ports from which channel content consumers read messages.

Figure 6 shows two components using an event channel. Note that there is no explicit connection between the content provider and consumer. This is the essence of an event channel. The Provides and Uses ports can be graphically located near to each other to suggest a connection, but the event channel is intended to separate the senders and receivers.

The generated XML for one of the connections in Figure 6 is shown in Figure 7. (The other is similar.) Like the other services, event channel connection ends are represented in SCA profiles as domainfinder findbys.

Once again, a convenient metaphor for the event channel freestanding ports is to view them as access points through the application layer to an event channel communication layer below.

Connections to Devices

The SCA is designed to make applications portable across different hardware platforms with minimal modification. Applications therefore have no direct knowledge of the platform on which they are deployed. However, applications must be able to use the platform through its logical devices. How can these connections be made in a portable way?

The SCA has two solutions to this problem: *devicethatloadedthiscomponentref* connections and *deviceusedbythiscomponentref* connections. Textual representations of software profiles can be challenging to understand and interpret. Freestanding ports clarify device connection specification, starting with devicethatloadedthiscomponentref connections then considering deviceusedbythiscomponentref connections.

Devicethatloadedthiscomponentref

An application architect knows the component instances that make up an application. A component instance is deployed or loaded onto a device when it is created, and this deployment mapping is available to the core framework to set up connections. A devicethatloadedthiscomponentref connection end specifies a component instance reference (i.e., a unique ID) and a port. The connection is literally made to the given port on the logical device whose physical device loaded the referenced component instance.

The word "this" in devicethatloadedthiscomponentref can be confusing to new SCA developers. Since the logical device port is usually (but not always) connected to a component instance, it's natural to assume that

```
<!--Connection between [Component generator1, Port Content_Out] and [Free-Standing
             Port Content_Channel]-->
<connectinterface id="DCE:e90686dc-c6ed-4a6f-97ae-478a7e6c0415">
    <usesport>
        <usesidentifier>Content_Out</usesidentifier>
        <componentinstantiationref refid="DCE:1fbbe2d8-cc5b-464f-be30-b3c159aa8092" />
    </usesport>
    <providesport>
        <providesidentifier>Content_Channel</providesidentifier>
        <findby>
            <domainfinder type="eventchannel" name="ContentChannel" />
        </findby>
    </providesport>
</connectinterface>
```

Figure 7

"thiscomponentref" refers to the connected component instance. However, any component instance can be used in the connection, not just the component instance at the other end of the connection.

The *SCA Developer's Guide* includes a graphical representation of a connection between a port on a component instance and port on a Modem device. The component instance and its port are shown in the conventional manner. The Modem device appears as a shaded component instance with a port. At first glance, this appears to be a reasonable way to represent the device end of the connection. However, it makes several implicit assumptions and causes representation problems.

The root problem behind this approach is that the application cannot and should not make assumptions about the device instance to which it is connected because doing so compromises portability. The device instance is part of the platform, not the application. The application should only know that the device instance has at least the port participating in the connection.

The solution is to recognize the independence of logical device ports from each other in the desired SCA representation and to represent the devicethatloadedthiscomponentref port as a freestanding port. The freestanding port's icon suggests the loading relationship between the referenced component and a device.

Figure 8 shows a connection between the Modem_Out port of a component instance and a freestanding Provides

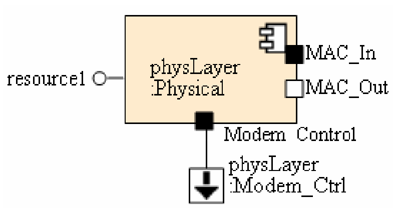

Figure 8

devicethatloadedthiscomponentref port. The devicethatloadedthiscomponentref port instance reference is specified as a component instance name instead of an unreadable unique ID. The SAD file XML that Figure 8 represents is shown in Figure 9.

Deviceusedbythiscomponentref

Devicethatloadedthiscomponentref gives one way of connecting to a logical device, but what happens when the logical device in question is not loadable? The SCA also provides a deviceusedbythiscomponentref connection from waveforms to logical devices. This connection type employs *usesdevice* dependencies which in turn make use of allocation properties, so we will start with a very brief overview of these.

An allocation property describes some resource provided by a component instance, and specifically a device instance. Properties can be qualitative or quantitative: a `bandwidth` property might describe the available bandwidth of a `modem` device that could be shared between component instances using the `modem`. In this `modem` example, the component instances sharing the modem don't execute on it.

Components specify their requirement for an allocation property with a *usesdevice* dependency that references the allocation property's unique ID. A component instance never directly specifies the device that it uses. The device provides an allocation property, the component instance has a usesdevice relationship on the property, and the Core Framework is responsible for connecting the two. A single component instance can use multiple devices through multiple *usesdevice* dependencies and a single device instance can provide one or several allocation properties to multiple component instances.

Deviceusedbythiscomponentref connections are the most complicated ones in the SCA, but with this background the remainder is straightforward. A connection end specifies a component instance and a usesdevice relationship on the definition that is the basis of that instance. Like devicethatloadedthiscomponentref connections, "this" specifies a component instance unique ID, not the component at the other end of the connection.

The device instance participating in the connection is not part of the portable application that uses it and the full interface of the device cannot and need not be known to the application. A natural representation of the connection does not show this unknown device instance.

Freestanding ports display exactly what is known, as shown in Figure 10. This example shows a Provides deviceusedbythiscomponentref freestanding port with its "U" icon. A Uses deviceusedbythiscomponentref freestanding port is displayed as a white "U" on a black square. The *thiscomponentref* and *usesdevice* information are properties of the freestanding port.

The connection in Figure 10 is equivalent to the SAD file XML in Figure 11. The deviceusedbythiscomponentref specifies the allocation property that identifies the device

```
<!--Connection between [Component physLayer, Port Modem_Control] and [Free-Standing
                    Port physLayer]-->
<connectinterface id="DCE:508963ae-eb30-43fd-858c-3e7f5b404846">
    <usesport>
        <usesidentifier>Modem_Control</usesidentifier>
        <componentinstantiationref refid="DCE:f8f96234-8672-4527-9afe-4dc77abe354b" />
    </usesport>
    <providesport>
        <providesidentifier>physLayer</providesidentifier>
        <devicethatloadedthiscomponentref refid="DCE:f8f96234-8672-4527-9afe-
4dc77abe354b" />
    </providesport>
</connectinterface>
```

Figure 9

```
<connectinterface id="DCE:3b63934c-edce-4f90-a8ba-370c072f1f2f">
    <usesport>
        <usesidentifier>Modem_Control</usesidentifier>
        <componentinstantiationref refid="DCE:1e37517d-7444-4ce8-b785-66f1fe4661d9" />
    </usesport>
    <providesport>
        <providesidentifier>ModemControl</providesidentifier>
        <deviceusedbythiscomponentref refid="DCE:1e37517d-7444-4ce8-b785-66f1fe4661d9"
                usesrefid="ModemBandwidth" />
    </providesport>
</connectinterface>
```

Figure 11

and also the component instance using the property. Here we see the situation where the thiscomponentref component is the component instance at the other end of the connection.

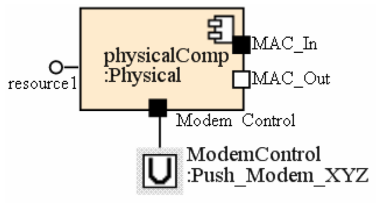

Figure 10

As a final note on connections to device instances, the architect of an application may expect that two deviceusedbythiscomponentref or devicethatloadedthiscomponentref connections will use the same device. The SCA does not provide any way to guarantee that this will happen on all deployment platforms. Freestanding ports therefore display exactly what the SCA can ensure. However, relationships between freestanding ports can still be captured. Graphical layout is a powerful tool for capturing architectural intent. To indicate that two deviceusedbythiscomponentref or devicethatloadedthiscomponentref freestanding ports share a device, place them together on an application assembly.

5. UML COMPLIANCE

SCA ports are closely related to the ports introduced to the Unified Modeling Language in UML 2.0. However, UML 2.0 "classes with structure" have no concept of a freestanding port. A port is part of a class or component, and can never exist on its own. However, freestanding ports are entirely compatible with an SCA profile of UML, and there is a simple mapping of freestanding ports to the ports of UML 2.0. "Freestanding" ports aren't really freestanding at all. They are a convenient representation of an aspect of a component that is visible in a given context. Every freestanding port belongs to a component; the component simply doesn't appear in a graphical view. By eliding the component, irrelevant and unknown aspects are hidden. The user of the freestanding port does not know or need to know how many other ports the unseen component may have.
If the idea of a port with no visible component is still worrying, simply draw a component around every freestanding port.

6. SUMMARY

Freestanding ports are a new tool for the SCA architect and developer. They are a valuable notation for all stakeholders who specify, design, implement, integrate, test or use reliable SCA applications. They simplify architectural design and communication even when the only tool available is a whiteboard. With tool support they can decrease the amount of deep SCA knowledge required to rapidly deliver a solid product. Freestanding ports are a powerful step forward in SCA modeling.

7. REFERENCES

[1] Joint Tactical Radio System (JTRS) Joint Program Office, *Software Communication Architecture Specification V3.0*, August 27, 2004. http://jtrs.army.mil/sections/technicalinformation/fset_technical_sca.html.

[2] Raytheon, *Joint Tactical Radio System (JTRS) SCA Developer's Guide*, June 18, 2002. http://jtrs.army.mil/sections/technicalinformation/fset_technical_sca.html.

[3] OMG, *UML 2.0 Superstructure Specification* (convenience document), October 8, 2004. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02.

[4] Hogg, John, *Communicating SCA Architectures by Visualizing SCA Connections*, 2005. http://www.zeligsoft.com/Technology/Resources.asp.