Supporting Component-based Development by Enriching the Traditional API

Marco Castaldi[1] and Nathan D. Ryan[2]

[1] Dipartimento di Informatica, Università degli Studi dell'Aquila
Via Vetoio, 1, 67010, Coppito, L'Aquila, Italy
castaldi@di.univaq.it
[2] Software Engineering Research Lab
Department of Computer Science, University of Colorado at Boulder
UCB 430, Boulder, CO, USA
ryannd@cs.colorado.edu

Abstract. Component-based software, particularly distributed component-based software, makes heavy use of interfaces to define the interactions among components. Each of these interfaces is usally presented in the form of an API, as a specification of the capabilities of the relevant component. However, while the predominant API model captures the behavior of individual interface members, it neglects the interactions among those members. We propose an approach that incorporates such interactions into an interface specification, and provide an application of the approach.

1 Introduction

An Application Programming Interface—whether for an object, a protocol, or an achitectural component—is normally expressed as, well, an interface, or in other words, a collection of functions (or methods) with perhaps some variables (or fields). These specifications are typically intended for automatic processing (as is the case with interfaces defined by systems such as the OMG IDL [7] and various programming languages such as Java. Often, each member of the API has an associated abstract, usually in the form of human-readable commentation, containing a description of the required behavior of each function and the intended semantics of each variable.

Such verbal descriptions are sufficient for human interpretation, and are certainly used to great effect. However, they are insufficient for automatic processing and a more formalized treatment of interfaces in general. Moreover, they are often the only source of description for interaction among interface members. For a component-based system, this "incompleteness" makes for more difficult development and validation.

2 A Problem with the Traditional API

Developers rely on interface member descriptions to understand just how to use an interface, since only in rare cases can the name of a function or variable convey

the appropriate semantics of its use, especially in relation to other members of the interface.

As a motivating example, consider a simple server interface that has two functions, `START` and `SUSPEND`. Intuitively, this interface definition indicates that `SUSPEND` should not be called before `START` (along with a couple of other obvious rules), or at least the implication exists. But without an accompanying description, we might not know the appropriate behavior. Does this sequence of calls cause an error, or throw an exception? Are either of these functions idempotent operations? Is an implementation of this interface in a "suspended" status before it is started for the first time? None of these questions are answered by the signatures of the two functions, unless some *de facto* standard exists.

While the informal description may not be necessary for a function or variable as it stands alone, clearly there are some relationships (constraints) among entities of an interface that are *only* captured by informal descriptions, even though they are likely simple enough to be represented formally.

2.1 Why Is This a Problem?

Simply stated, one can keep only so much in one's head. Any developer would be able to understand a simple, isolated interface like the one in the example above. But what if, in a large component-based system, you have hundreds of interfaces, each with dozens of entities? Now the problem is significantly more complicated.

An additional problem arises when the notion of automation is introduced. With only informal descriptions, automatic enforcement of the relationships among the members of an interface is impossible. For more dynamically-oriented systems, such as a reconfigurable component-based system, the ability to automatically validate internal interface constraints that are possibly different or even unknown at the start of a run could be of great use.

3 A Solution for the Traditional API

The solution seems an obvious one: Formalize these informal descriptions, and include them as an integral part of the API. This has already been introduced in approaches such as Path Expressions [3] and Design by Contract [5], and in practical tools that allow one to easily specify validations within the code itself, such as APP for C [4] and the powerful assertion mechanism within Eiffel, among many others.

However, the relationships among the members of an interface should be specified within the interface on a global basis, separate from the realizations of those members. In other words, we would like an interface specification to include a section or sections (apart from the member declarations) for *all* of the relationships among those members.

For the example in the previous section, a path expression such as

```
path START ; SUSPEND end
```

may somehow need to be expressed. For an interface without variables (or in other words, for an interface that makes no assumptions about the statefulness of an implementation), path expressions are often enough; however, if a variable such as `MASTER` is introduced to our example to capture a parent server's address, several additional constraints may also be required. Regardless, it is useful to capture these relationships as a whole, and bundle them with the interface in order to provide not only the appearance of an implementation but also the *behavior* required by the interface.

3.1 Why Is This a Solution?

Foremost, it allows the previously informal constraints to be formally expressed. This concept is nothing new, however; countless formalisms exist to describe such constraints.

But such an enriched interface has additional benefits. For example, if several functions have the same (possibly large) set of preconditions, these can be captured as a group and applied to each function as a group. As another example, relationships of implication among variables can be expressed, something difficult to do without a pair of "get" and "set" functions specific to each variable, which is something not always desirable.

However, the most important benefit is that a mechanism for validating the constraints of an interface can be automatically generated and applied to *any* realization of the interface, even apart from the implementation. This is especially convenient for distributed component-based systems, that conceptually deal exclusively with interfaces via a management protocol such as SNMP [1]. With such an interface (and proper coordination among managing components, of course), a requesting component could locally validate an action on another component, even without the receiving component needing to process the request. Enforcing constraints at the time of the operation but prior to execution (whether locally or remotely) not only isolates the components from some of the duties of error reporting but can possibly prevent components from entering an inconsistent state due to an invalid request. Furthermore, the same mechanism can be applied to *all* components that implement the interface.

4 A Practical Example

To demonstrate the problems with a traditional interface description and introduce a possible solution, we will extend the simple example above using LIRA (Light-weight Infrastructure for Reconfiguring Applications) [2], a system for remote management via dynamic and automatic reconfiguration of component-based applications in the context of large-scale distributed systems.

LIRA uses and extends the approach of network-management architectures and protocols, where an Agent directly controls the managed device and a Manager initiates the reconfigurations.

Here we further describe a LIRA Agent for a server in Siena (Scalable Internet Event-Notification Architecture) [6], a general publish/subscribe service that uses an innovative content-based routing mechanism.

4.1 LIRA

LIRA furnishes a general interface to act over components and applications. This interface is realized using a simple architecture and protocol, allowing one *to "get" and "set" variables* and *to call functions*.

The architecture uses three main actors: the **Reconfiguration Agent**, which contains the knowledge necessary to reconfigure the component (or system); the **Component Description**, which lists the functions and variables that the Agent exports in order to enable a reconfiguration of the component; and the **Management Protocol**, which allows Agents to communicate with each other.

To support a hierarchy of Agents, in which a higher-level Agent performs a reconfiguration at the application level using functions and variables exported by lower-level Agents, LIRA Agents can act in two different roles: an **Actuator**, which receives the reconfiguration order and performs the reconfiguration on the components; and a **Manager**, which gives orders to an Actuator regarding the reconfiguration to be performed on the components.

The Management Protocol is a simple TCP/IP connection-based protocol with six synchronous messages and one asynchronous message: GET/REPLY and SET/ACK (for "getting" and "setting" variables), CALL/RETURN (for function calls), and NOTIFY (to notify an Agent of some specific event.)

4.2 A Reconfiguration Agent for the Siena Server

The reconfiguration we want to perform on the Siena Server is to change the server's master, represented by the parameter `MASTER`, which binds the various Siena Servers together in order to create a Siena Network. If server S_1 is a master for servers S_2 and S_3, a message sent from a client attached to S_2 may reach another client attached to S_3, but only through S_1. If S_1 is down, S_2 and S_3 are isolated from each other.

To dynamically change a server's master, the LIRA Agent managing the server exports the variable `MASTER`. A portion of the Component Description (using a traditional interface) for the Siena Server follows:

```
Variables {
  MASTER {
    type = String;
    description = "The address
     as 'senp://host:port'.
     Cannot be changed without
```

```
    losing subscriptions unless
    the server is suspended.";
  }
  ...
}
Functions {
  START() {
    parameters {};
    return = "a message with
     the result of the start";
  }
  ...
}
```

In the `description` field of `MASTER`, an important constraint is specified: that the master "cannot be changed without losing subscriptions unless the server is suspended". (In fact the Siena Server has an internal mechanism that saves subscriptions when the server is suspended.)

Note that an automated manager can change the master and unknowningly loose information; this is a consequence of the informal interface specification, which does not allow an automated system to utilize crucial information furnished by the component developer. This is simply a demonstration of the problem with the traditional API, as discussed earlier.

A solution adopted for the LIRA Agent is to enrich the Component Description by specifying a list of **Constraints** and **Actions**. A Constraint is of the form

```
message => condition
```

where the operation associated with *message* is executed only if the boolean *condition* expression is satisfied. An Action is of the form

```
message (-> result)+
```

where the boolean *result* expressions successively occur by sending the specified *message*.

For the Component Description of the Siena Server, we add the following:

```
Constraints {
  SET(MASTER, x)
    => STATUS == suspended;
}
Actions {
  CALL(SUSPEND, void)
    -> STATUS == suspending
    -> STATUS == suspended;
}
```

With this approach the constraints can be formally specified, allowing an automated manager to determine the conditions under which some messages may be sent, and what actions must be performed in order to satisfy those conditions.

5 Conclusion

Obviously, the example presented here is a simple one. Our intention was merely to give a concrete demonstration of the approach; how it is generalized and implemented has implications for wider applicability.

Regardless, by using this approach we can formalize constraints that normally are expressed verbally. When the relationships of an interface are formalized, not only can they can be automatically validated, they can be validated across all implementations of the interface, independent of the implementations. Further, since the contract defined by the interface is more explicit, the behavior of components implementing the interface is better understood, thus assisting the development of component-based software.

References

1. Marshall T. Rose: "The Simple Book: An Introduction to Networking Management", Prentice Hall, April 1996.
2. Marco Castaldi: "LIRA: Light-weight Infrastructure for Reconfiguring Applications", Technical Report, Department of Computer Science, University of Colorado at Boulder.
3. R. H. Campbell and A. N. Habermann: "The Specification of Process Synchronization by Path Expressions", in Operating Systems, International Symposium, Rocquencourt, vol. 16, Lecture Notes in Computer Science, 1974, pp. 89-102.
4. David S. Rosenblum: "A Practical Approach to Programming with Assertions", IEEE Transactions on Software Engineering, vol. 21, no. 1, Jan. 1995, pp. 19-31.
5. Bertrand Meyer: "Applying 'Design by Contract' ", IEEE Computer, vol. 25, no. 10, October 1992, pages 40-51.
6. Antonio Carzaniga: "Architectures for an Event Notification Service Scalable to Wide-area Networks", PhD Thesis, Politecnico di Milano, December, 1998.
7. "OMG IDL Syntax and Semantics", OMG CORBA v2.6 specification, December 2001.