

Compatibility vs. Evolution in Pervasive Computing

Position paper for Building Software for Pervasive Computing '04
(OOPSLA '04 Workshop)

Emanuela P. Lins and Ulrik P. Shultz
DAIMI, Aarhus University
{emanuela,ups}@daimi.au.dk

August 20, 2004

1 Introduction

The emerging trend of pervasive computing is giving rise to a wide variety of networked, “intelligent” home devices, including AV equipment, standard domestic appliances such as refrigerators, and various small sensors and actuators. To be truly useful, these devices should as a minimum be able to exchange information and make use of each other’s services. In other words, they should be compatible. Compatibility is not easy to achieve for this kind of environment, in particular since new devices with new and updated software are introduced over time. Indeed, we consider it likely that compatibility must be maintained over many years: users are not likely to replace their domestic appliances at the same rate as their personal computers, so new devices must take into account numerous older generations of software embedded into devices dispersed throughout the domestic environment. Thus, distributed interface compatibility and evolution is a fundamental issue in pervasive computing.

To maintain compatibility, each new generation of devices must either restrict the evolution of their distributed interface or implement adapters for each older version. Alternatively, dynamic software update for the older devices provides a solution to this issue, but carries its own set of problems. First, dynamic software updates complicate testing, since every hardware and software combination must be

tested in advance to avoid device malfunctioning. Second, the ability to update the software dynamically is likely to increase the manufacturing cost for e.g. mass-produced low-end sensor systems designed to be spread throughout the house. Last, updating one device in an incompatible way can cause other devices which have not yet been updated to fail — the complexity of transparently updating software in a distributed, embedded system is likely to defeat most home users.

This paper gives an overview of the design and implementation of an interface declaration language that resolves the evolution versus compatibility tension for distributed objects by allowing the programmer to declare how newer versions of distributed objects evolve from older versions, and then using these declarations to automatically generate adapters. We focus on distributed communication between digital devices connected by a digital network, where the communication is performed using a remote method invocation (RMI) protocol.

1.1 Existing technologies

Before we describe our own solution, we briefly review three state-of-the-art technologies, namely CORBA, Java RMI, and XML.

CORBA Subclass relationships between interfaces is typically used to provide rudimentary sup-

port for versioning. The CORBA approach of using *struct*-types for exchanging data greatly simplifies distributed communication, but also implies design-level problems when using object-oriented languages. Data structures that are to be exchanged between distributed objects must either be written using simple structs or be manually converted into structs and then used to build new object graphs on the other side of the distribution barrier.

Java RMI One of the advantages of Java RMI is that it has value objects which we consider to be a powerful and highly useful feature. There is no need to use structs for distributed communication. Apart from the presence of remote exceptions, accessing remote objects works as if it was done locally, the distributed communication is transparent to the programmer. On the other hand, a major disadvantage of Java RMI with respect to remote interface evolution is that Java RMI does not allow renaming or removing remote methods (it is only allowed to add new ones), and most changes to fields or refactorings of the class hierarchy implies manual work by the programmer to maintain compatibility, if it is at all possible.

XML With respect to communication between original and evolved devices, XML could probably be used for data conversion in most cases. Nevertheless, the time and space overhead due to encoding/decoding, transmission, and parsing in XML could be a limitation, since devices might only be equipped with limited processing capability suited to the task at hand. Moreover, conversions between versioned devices, presumably done using XSLT, still have to be implemented manually by the programmer rather than generated automatically based on the distributed interface of each device.

We consider that the language-neutral IDL-based approach of CORBA is useful, but that value objects are an essential feature that must be handled by our solution. We assume that the manufacturer of a device

releases IDL specifications for each version of the device, which allows other manufacturers to interface with the device. Moreover, we focus on resource-constrained devices, which makes it relevant to exclude dynamic software updates (including dynamic class loading) and XML-like generic solutions.

1.2 Our approach

We observe that compatibility is a cross-cutting concern in distributed systems, and that this concern is particularly acute in the case of pervasive computing systems: new devices must address compatibility with all versions of software found in existing devices already situated in the users environment. To express compatibility, we use a domain-specific extension to a standard IDL-language. This language, named “Versioning Interface Definition Language” (VIDL), allows declaring compatibility-specific functionality, such as conversion rules, and gets these declarations out of the way of the main implementation. The main idea is to evolve software using refactorings while allowing older versions of the software to coexist with the newest versions by making it possible for them to communicate. VIDL enables the programmer to declare versioned devices which are compatible (at the interface level) and can communicate with each other by exchanging objects via RMI.

2 VIDL

The VIDL language allows transparent evolution of distributed objects. VIDL supports maintaining compatibility between distributed devices across a large selection of standard refactorings [1], such as e.g. changing a concrete class by making it abstract (and vice-versa), evolving classes by adding, changing and removing fields, and evolving interfaces by adding, changing and removing methods. These refactorings should in many cases be sufficient for ensuring freedom to evolve the implementation of each device. The VIDL compiler automatically generates the code for supporting distributed communication between such evolved devices.

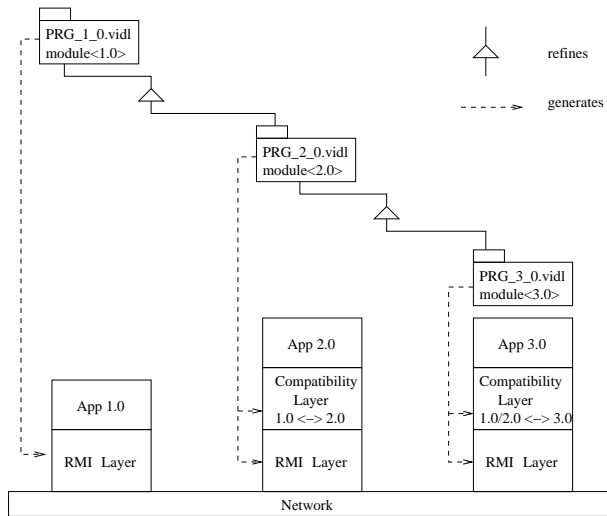


Figure 1: Diagram of a VIDL-based specification.

2.1 Basic concepts of VIDL

A VIDL specification declares the distributed interface of one or more modules. A *VIDL module* represents a package and contains all information needed about a specific version such as classes, interfaces with their methods and conversion rules for distributed communication. A *VIDL class* declares the serialized format (fields) of a class as well as the interface it implements. A class can be concrete or abstract: a concrete class can be instantiated, whereas an abstract class cannot. A *VIDL interface* is a declaration of methods through which a remote object can be accessed.

Evolution of components is declared in VIDL through refinements, as illustrated in Figure 1. A *VIDL refinement* represents a component's evolution. A module m' refines a module m when m' evolves from m . It is only necessary to declare what has changed and the rules to communicate with m . Based on the refinements, the VIDL compiler generates adapter code for every previous version of the components, which enables distributed communication between the versioned devices.

VIDL addresses compatibility between different

versions via conversion rules for distributed communication. A *VIDL conversion rule* can be a sending rule or a receiving rule. The former is a conversion rule to be applied when making a remote method invocation; it concerns the signature of the called method as well as the objects (if any) that are passed as arguments of the method call. The latter is a conversion rule to be applied when receiving a remote method invocation or an object returned by a previous remote method invocation. The VIDL syntax in general and conversion rules in particular are illustrated by the following example. For details, we refer to the first author's MS [3].

2.2 Example

As an example, consider a television set which communicates with a (large) number of small speakers dispersed throughout the living room. Each speaker is an independent device, that can be remotely controlled over the network. Figure 2 shows part of the software; it concerns configuring sound. Assume that this is the version of the software shipped with the first release of the devices. There is a remote interface for each speaker through which the speaker can be remotely controlled and the sound features of each speaker such as volume, bass and treble levels can be stored. Note that the VIDL source code has been manually pretty-printed with a slightly more standard syntax and that the keywords are written in bold.

For the next release of the speaker and television software, a number of improvements and enhancements are made to the public interface. A new method has been added to mute the sound. Moreover, sound can be configured either as before or in terms of a number of pre-defined profiles. Last, the methods `powerOn()` and `powerOff()` have been removed and the parameterized method `setPower()` has been added instead. The evolved version of the software is shown in Figure 3.

Conversion rules are used to specify how the new version of the module can communicate with the previous version. A sending rule is declared as `to:<version>`, where `<version>` normally (but not necessarily) is the latest previous version.

```

module spk<1.0> {
  import sound<1.0>

  new remote interface Speaker {
    new void setSoundConf(SoundConf cfg)
    new SoundConf getSoundConf()
    new void powerOn()
    new void powerOff()
    new void setStreamingChannel(Channel ch)
  }

  // ...
}

module sound<1.0> {

  new SoundConf {
    int volLevel, basLevel, trebleLevel;
    // ... methods
  }

  // ...
}

```

Figure 2: Version 1.0 of the speaker interface.

All receiving rules are declared inside the clause `from:<version>`, where `<version>` again normally is the latest previous version. Inside the rule, whenever the symbol “\$” precedes the name of a field n , it means that the value denoted by the field n is to be used instead. Note that in a receiving rule, this is a field in the previous version. In the module `Speaker`, conversion rules are used to map between the version 1.0 methods `powerOn` and `powerOff` and the version 2.0 method `setPower`. As can be seen, conversion rules can contain a pattern as criterion for when the rule should be applied. Currently, VIDL only supports pattern matching based on constants. More advanced pattern matching is future work. The declaration of a pattern matching in VIDL uses the name of the field (which stores the target value) followed by “==” and the name of the matching constant. For example: “`isOn==SoundConf<2.0>.ON`”. Note that for each new method there is a sending rule for the previous version. Moreover, for each removed method there is a receiving rule for the previous version.

The class hierarchy defined in the module `sound`

```

module spk<2.0> refines spk<1.0> {

  change remote interface Speaker {
    remove void powerOn()
    remove void powerOff()
    new void setPower(int isOn)
      to: 1.0 $isOn==SoundConf<2.0>.ON
          => void powerOn();
      to: 1.0 $isOn==SoundConf<2.0>.OFF
          => void powerOff();
    new void setMute(boolean isMuted)
      to: 1.0 =>
          void setSoundConf(
            @getMuteConf($isMuted)
          );
  }

  from: 1.0 {
    void powerOn() =>
      setPower(isOn=SoundConf<2.0>.ON);
    void powerOff() =>
      setPower(isOn=SoundConf<2.0>.OFF);
  }
}

module sound<2.0> refines sound<1.0> {

  change abstract SoundConf { int OFF=0, ON=1; }

  new AbsoluteSoundConf extends SoundConf {
    int volLevel, basLevel, trebLevel, toneLevel; }
    to: 1.0 => SoundConf<1.0>
        (volLevel=$volLevel,
         basLevel=$basLevel,
         trebLevel=$trebLevel);

  new ProfileSoundConf extends SoundConf {
    int profileID; }
    to: 1.0 => SoundConf<1.0>
        (@getProfileVol($profileID),
         @getProfileBas($profileID),
         @getProfileTreb($profileID));

  from: 1.0 {
    SoundConf<1.0>(volLevel,basLevel,trebLevel)
      => AbsoluteSoundConf(volLevel=$volLevel,
                           basLevel=$basLevel,
                           trebLevel=$trebLevel,
                           toneLevel=@getDefaultTone());
  }
}

```

Figure 3: Version 2.0 of the speaker interface.

is refactored by making the class `SoundConf` be abstract with two concrete subclasses, `AbsoluteSoundConf` which corresponds to the original class `SoundConf` except that a new field has been added, and the new class `ProfileSoundConf`. Conversion rules are used to map objects of these classes to and from the former version during serialization, when communicating with an older device.

In some cases the programmer might need to call external code in order to write appropriate conversion rules. For example, in Figure 3 when a device version 2.0 calls the method `setMute()` on a device version 1.0, the external method `getMuteConf` is called to compute a (de)muted configuration based on the current configuration. External methods are prefixed by “@” and must be written in the language targeted by the VIDL compiler (currently Java but might as well be e.g. C++).

3 Implementation

Our solution is based on the Audio-Video Software (AVS) middleware used in the B&O+OO research project which concerns the development of a complete software and hardware platform for next-generation AV devices for the Danish AV-systems producer Bang&Olufsen. AVS is similar to Java RMI in terms of calling semantics and serialization of objects passed as arguments, but is much less advanced in terms of features, which makes it easy to modify in order to interface with the VIDL-generated code.¹ The current implementation of the VIDL compiler generates all the code needed for distributed communication between versioned devices such as those shown in the example. See [3] for a larger, more realistic example.

Complex conversions The devices communicate by means of exchanging objects and applying conver-

¹An initial case study done using Java RMI showed that it was surprisingly difficult to implement some of the features currently supported by VIDL. For example, a concrete class cannot easily be evolved into an abstract one using the means provided through the standard Java RMI interfaces `Serializable` and `Externalizable`.

sion rules. Usually VIDL generates all code for distributed communication automatically. There are, however, some complex cases for which this is not possible. In this case, VIDL can delegate the conversion to a programmer-written handler which must simply implement the appropriate handler interface. A complex conversion rule is both context-dependent in the sense that it can depend on what method is being called and it allows local computations to be performed (before sending or just after receiving an object).

Optimizations The VIDL language allows declaration of distributed-versioned interfaces of devices based on user-declared conversion rules and incremental refinement by version. This means that, as the specification evolves, the programmer only needs to declare conversion rules for the latest previous version. Nevertheless, conversions between versions with many intermediate changes could easily result in an inefficient implementation, for which reason the VIDL compiler internally reduces the conversion rules to a single, complete rule for each version. This is done by composing the conversion rules which are declared in the VIDL specification, similarly to deforestation [5].

Static verification of compatibility VIDL allows the programmer to describe evolution of module versions. During the compilation process, the compiler checks the consistency of the program. When a VIDL declaration is compiled by the VIDL compiler and the automatically generated code is compiled by the Java compiler, the devices with distributed-versioned interfaces described in the VIDL specification are compatible by construction. For example, changes to the method signature must be handled by conversion rules, as must changes to the object layout.

4 Open issues

For a given component, the VIDL specification expresses not only the current interface but also the

interfaces of all prior versions, including the conversions that are needed to maintain compatibility. In effect, compatibility with older versions is an intrinsic part of the interface. Nevertheless, the conversions could be separated from the interface as a different aspect, which would likely improve readability of the specification. However, it is not clear how the conversion rules should then be declared nor if there are other, separate concerns that are also relevant to the issues addressed by VIDL.

A major weakness of the VIDL approach is that it only offers compatibility guarantees at the interface level; we offer no support for ensuring semantic compatibility. We believe that the introduction of pre/post conditions associated with the method declarations could help enforce compatibility at the semantic level, but we are still investigating this issue.

References

- [1] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [2] N. Lesiecki. Improve modularity with aspect-oriented programming, January 2002. URL: <http://www-106.ibm.com/developerworks/library/j-aspectj/>.
- [3] E. P. Lins. Evolution, Versioning and Compatibility of Distributed Objects. Master's thesis, Aarhus University (DAIMI), June 2004. URL: <http://www.daimi.au.dk/~emanuela/thesis.pdf>.
- [4] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, second edition, 1997.
- [5] P. Wadler. Deforestation: Transforming programs to eliminate trees. volume 73(2), pages 231–248. Theoretical Computer Science, June 1990. Special issue on ESOP'88, the Second European Symposium on Programming, Nancy, France, March 21-24, 1988.