Supporting Transparent Evolution of Component Interfaces[*]

Emanuela P. Lins[†] and Ulrik P. Schultz[‡]
ISIS, DAIMI, University of Aarhus
{emanuela,ups}@daimi.au.dk

ABSTRACT

Component-oriented programming facilitates the development of reusable application parts encapsulated by well-defined interfaces. There is however a tension between compatibility and evolution, since the interface of a component may constrain refactoring or require manual development of multiple, ad-hoc adaptation layers when an interface is evolved. We here present the declarative language VIDL for specifying component interface evolution. VIDL allows evolution of components with automatic generation of efficient adapter code that statically guarantees interface compatibility with other components that rely on anterior versions of the interface.

1. INTRODUCTION

The emerging trend of pervasive computing is giving rise to a wide variety of networked, "intelligent" home devices. To realize the full potential of pervasive computing, devices must however be able to communicate, meaning that they should be compatible. To maintain compatibility over time, 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 remedy, but carries its own set of problems in terms of testing, increased costs, and additional complexity.

To address the compatibility issue, we have developed a domain-specific IDL-language named "Versioning Interface Definition Language" (VIDL). VIDL allows declaring compatibility-specific functionality for transparently connecting components that communicate using different versions of the same interface. The main idea is to both allow software evolution using standard object-oriented refactorings [3] and allow older versions of the software to coexist with the newest versions by making it possible for them to communicate.
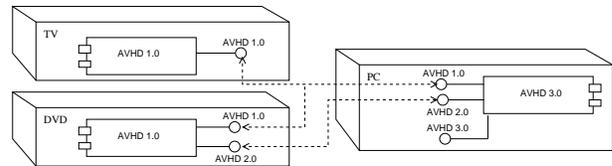

Figure 1: Deployment of multiple software versions

2. VIDL

VIDL describes the distributed interface of a software component, including rules for translating requests to and from older versions of the interface. Thus, a VIDL-generated component can at run-time be regarded by other distributed components as having multiple interface versions. Deployment of multiple component versions is illustrated for an AV harddisk (AVHD) component in Figure 1, where automatically generated adapters allow components to communicate using the newest supported interface version.

A VIDL specification is separated into modules that describe specific versions of the component interface. Newer, evolved versions of the same interface are created by *refining* a module with another module. It is only necessary to declare changes and corresponding conversion rules to translate requests between versions: the compiler verifies that the conversion rules are complete (e.g., cover all cases) and generates the required adapter code, as shown in Figure 2.

In more detail, a VIDL module contains classes, interfaces, and rules for distributed communication with older module versions. Each module declaration is annotated with a major and minor version. 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 (abstract classes cannot be instantiated). An interface is a declaration of methods through which a remote object can be accessed. Compared to CORBA IDL, VIDL is a subset of CORBA IDL (with value objects), extended with support for ver-
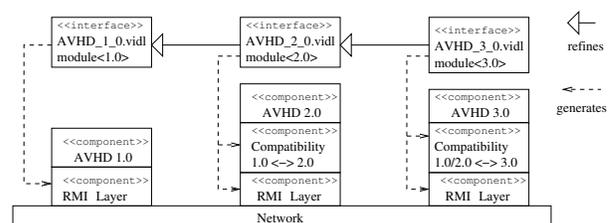

Figure 2: VIDL module refinement

```
module Clock<1.0> {
 new class Time { int h, m, s; }

 new remote interface Clock {
  void setClock(Time t);
  Time getTime();
 }
}
```

Figure 3: Version 1.0 of the module `Clock`.

```
module Clock<2.0> refines Clock<1.0> {
 change abstract class Time { int h, m, s; }
 from(1.0) { Time<1.0>(h,m,s) =>
             LocalTime(h=$h,m=$m,s=$s,tz=0) }

 new class LocalTime extends Time { int tz; }
 to(1.0) => Time<1.0>(h=$h,m=$m,s=$s)

 new class Alarm extends Time { int volume; }
 to(1.0) => throw OperationNotSupportedException()

 change remote interface Clock {
  new void setTime(Time t) to(1.0) => setClock(t)
  remove void setClock(Time t) from(1.0) => setTime(t)
 }
}
```

Figure 4: Version 2.0 of the module `Clock`.

sioning and automatic conversion between versions.

As an example, consider the clock component interface shown in Figure 3 for setting and getting the current time. All declarations are prefixed with the keyword `new` (VIDL requires explicitly specifying whether a declaration is new or modifies an older one). For the next public release, a time-zone and an alarm functionality has been added, as shown in the refinement module of Figure 4. The value object `Time` has been refactored as an abstract class with two concrete subclasses, and the operation `setClock` has been renamed to `setTime`. In all cases, conversion rules specify how to make newer devices communicate with an older clock.

A conversion rule can be attached to a method to specify how to translate calls back and forth between the current and the previous version. Similarly, a conversion rule can also be attached to a value object to specify how to translate the instance between versions (object identity is preserved since translation is only performed for distributed calls with a call-by-value semantics). Nevertheless, in some cases, conversions are more complex than the value mappings supported by the standard VIDL language constructs, for which reason VIDL allows the programmer to call external functions within the conversion rules or delegate value object conversions to a user-defined handler. For details on VIDL semantics, we refer to the first author's MS [4].

3. COMPILER OPTIMIZATIONS

The VIDL compiler runs in three phases: parsing, normalization, and code generation. The normalization phase propagates declarations from superclasses to subclasses and from older module versions to newer module versions, in effect flattening both the inheritance hierarchy and the module refinement hierarchy. For each module, dedicated conversion rules are generated for all anterior module versions by composing conversion rules. When there are different conversion rules declared for the same method in differently versioned modules of a VIDL specification, the normalizer composes the declared conversion rules associated with this method and generates a single conversion rule for the whole chain of method conversion rules. This makes the serialization layer more efficient because we apply only one conversion rule at runtime. Similar optimizations are performed for value object conversion rules. We observe that this optimization is similar to deforestation, in that intermediate conversion results are eliminated, but more limited since we can only optimize code within the serialization layer independently on each device [9].

4. RELATED WORK

The use of interface adapters to bridge components of different versions is a standard technique in component-oriented programming [6]. Nevertheless, we are not aware of any related work where interface adapters are generated statically as part of the serialization layer. Dynamic updating of software components is addressed both by Bialek and Jul and by Vandewoude and Berbers [1, 7], but there is no direct language support for specifying the conversion rules for interface adapters, and object structures cannot be automatically translated in the serialization layer which imposes an extra overhead at runtime. Vandewoude and Berbers describe the design of a tool for automatically deriving conversion rules [8]; VIDL could be used to specify conversion rules not automatically derived. Plasil et al present an architecture for dynamic component updating [5], but the main focus is on managing components, and component updating is only allowed when the replacement interface is a subtype of the installed interface. Duggan relies on a type system to automatically drive on-demand conversions between different versions of components [2], thus defining a flexible dynamic update mechanism for a core calculus that can be generalized to more complete languages.

5. REFERENCES

[1] R. Bialek and E. Jul. A framework for evolutionary, dynamically, updatable, component based systems. *ICDCS 2004 Workshops — DARES*, pages 326–331, March 2004.

[2] D. Duggan. Type-based hot swapping of running modules. *ICFP 2001*, pages 62–73, 2001.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[4] E.P. Lins. Evolution, versioning and compatibility of distributed objects. MS, Univ. of Aarhus, May 2004.

[5] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. *ICCDS'98*, May 1998. IEEE CS Press.

[6] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 2002.

[7] Y. Vandewoude and Y. Berbers. Supporting runtime evolution in SEESCOA. *Journal of Integrated Design & Process Science: Transactions of the SDPS*, 8(1):77–89, March 2004.

[8] Y. Vandewoude and Y. Berbers. Fresco: Flexible and reliable evolution system for components. *Electronic Notes in TCS*, 127(3):197–205, April 2005.

[9] P. Wadler. Deforestation: transforming programs to eliminate trees. *TCS*, 73:231–248, 1990.