NSI: A Component Model for Consumer Electronics Software

Abstract

Component-based software engineering (CBSE) is becoming widely practiced in the desktop and enterprise computing arenas. Ironically, despite its applicability, the many benefits of CBSE have been largely unavailable to Consumer Electronics software developers. In addition, the Consumer Electronics (CE) industry has seen a revolution over the past decade, where CE manufacturers have changed from hardware manufacturers who ship a little software, to software providers who ship generic digital computing devices, with speciailized I/O peripherals and in a specialized form factor.

NSI (Nexwave Software Infrastructure) is a software component model targeted at embedded software development, and specifically at CE software. The NSI component model is a simple binary model, with associated tools and runtime environment. The components contain native code in order to provide the power and efficiency necessary for CE software. NSI interfaces are simple yet powerful mechanisms by which the often complicated interdependencies between components can be managed. NSI has been designed to allow easy conversion of existing monolithic or modular code into components.

This paper presents briefly CE's need for a new component model, a high-level description of the NSI model, and some experiences of its use by some of the major Japanese CE manufacturers.

1 Introduction

Component technology has been widely deployed over the past decade in the desktop and enterprise software arenas. Technologies such as Java [1], COM [2] and .NET [3] all bring real benefits to programmers. However, despite CBSE's striking applicability to Consumer Electronics (CE) software, none [4] of the myriad component models available provides a suitable platform for the majority of CE software.

NSI (Nexwave Software Infrastructure) is the first component model that we are aware of that is available to CE manufacturers and explicitly addresses their needs. In this paper we start by introducing briefly the concept of Component-Based Software Engineering (CBSE), and why it is particularly applicable to CE software, especially legacy code. Section 2 presents a high-level overview of the NSI component model. The NSI runtime environment is described in Section 3; and off-line tools for use with NSI components in Section 4.

1.1 What is Component-Based Software Engineering?

The question what is component-based software engineering is difficult to answer succinctly: there are almost as many different definitions of CBSE as there are component systems. Broadly speaking, CBSE is the technique of breaking software down into reusable components, and then composing software systems by joining together different combinations of components. Developing software in this fashion promises many advantages to the CE industry (explored in [4]); the most relevant being software reuse across product generations and product ranges.

One of the more widely accepted definitions of a software component is Szyperski's:

> A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [5].

A component's interface defines the syntax and semantics of how the implementation behind it is accessed. Implementations are separated from interfaces, in that implementations on either side of an interface know and care only about the interface, not the other implementation.

Components are independently deployed, meaning that components from different authors can be assembled by a third-party, without any explicit cooperation between any of the parties.

1.2 CE's need for a new component model

We believe that CE has a particularly urgent need for more effective software reuse, particularly across product ranges (e.g., between a digital cam-corder and a PVR), and that CBSE is the best way to achieve this. In our experience, CE manufacturers are increasingly suffering under the huge burden imposed by the development of the software that controls their devices. Consumers are demanding increasingly advanced functionality, and an increasing proportion of that functionality is provided by software (as opposed to hardware).

Ten years ago, it was practical to write from scratch the software for each CE product (or at least each product family). This has become increasingly less acceptable and is beginning to become infeasible. Instead, CE manufacturers must find a way to share software across their product range. Furthermore, it is becoming increasingly desirable that CE companies are able to share software between each other. Different manufacturers rewrite from scratch each time large amounts of software between themselves, regardless of how effectively individual manufacturers manage code reuse within their organization. Much of this software offers no product differentiation or competitive advantage to the CE manufacturers, and thus would more effectively be shared between them. (According to CE manufacturers, up to

80% of their code is "non-differentiating"—i.e., code that is broadly the same for all manufacturers, and does not influence the end consumers' experience, or their choice of which product to buy.)

There are several component models currently available and in wide use, but none is well suited to CE software (or embedded software generally). Widely known component technologies include CORBA [5], Java [1], COM [2] and .NET [3]. Very briefly: the CORBA Component Model imposes overheads typically unaccepted to CE vendors, and is targeted towards enterprise systems[1]; Java and .NET are "virtual machine" models, great for higher-level application software, but poorly suited to the low-level systems software that tends to dominate CE software development; and .NET and COM are both tightly linked to Microsoft platforms.

These arguments are inevitably controversial, and warrant further explanation and justification. A complete analysis of the suitability of CBSE to the CE industry, along with why existing component solutions are inappropriate, is beyond the scope of this paper; such analysis can be found in [4].

2 NSI: Components for CE software

NSI is a component model with supporting tools, designed for embedded software and specifically targeting the CE industry. Each NSI component consists of one or more interfaces and one implementation. An NSI component is a stand-alone binary, containing its implementation and a description of its interfaces. The build process for an NSI component is shown in Figure 1.

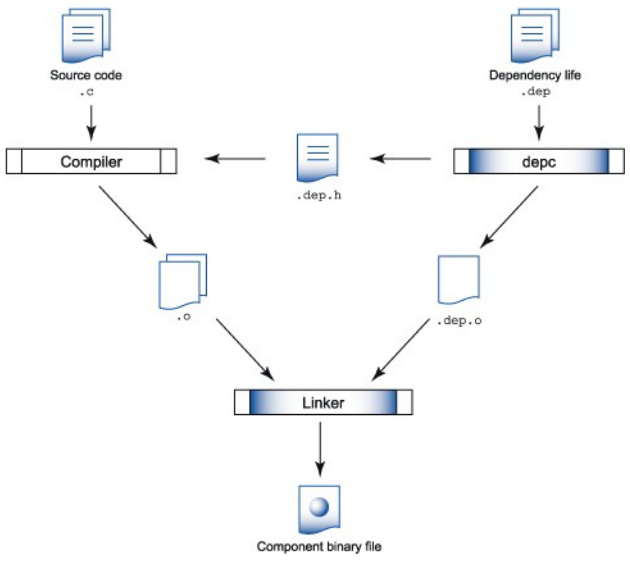

Figure 1: Files and tools in the building of an NSI component

1. Despite talk of a "Lightweight CORBA Component Model" [7], we are aware of no commercially available implementations of this.

Each NSI component is built from one interface definition file (the dependency file), and zero[2] or more C, C++ or assembler files. In this sense, the only differences between an NSI component and a traditional object/library file are the following:

- The interface description is injected into a special section.
- There are no undefined references or exported symbols, ensuring that all interaction with other components occurs via explicitly defined interfaces.

That is, at the binary/component level, NSI is a way to add interfaces support to object files (although using a somewhat new way to describe interfaces; see Section 2.1). In addition however, NSI defines a runtime environment in which components execute (see Section 3), along with supporting tools for building component-based systems (Section 4).

2.1 NSI Interfaces

Inter-component communication in NSI occurs between interfaces. An interface consists of a name, and a set of functions and read-only data attributes. Interfaces can be exported (provided) or imported (requested). Two components communicate over an interface through an import link. The import link will only be created if the requested functions and attributes are provided by the exporting component. The import link is type-safe, meaning that the exporting and importing components agree on the type signatures of the interfaces' functions and attributes.

An interface's name consists of two parts: a human-friendly part, and a unique part. The human-friendly name is a textual identifier (e.g., `video_out`), while the unique part is a 128-bit UUID. Exported and imported interface names must be identical (both the human-friendly and unique parts). The NSI model itself encodes only syntax in interfaces. An interface's name is effectively a link to the specification of its semantics, which is usually defined in an external document. NSI interfaces are defined in the component's dependency file, as opposed to declaring the interface centrally. The dependency file is written using NexWave's Dep language, which is based on OSF DCE IDL [8]. The NSI dependency compiler is designed to be used with C types. This means that programmers can simply `#include` a C header file, and then use the relevant types in their interface definition (as opposed to needing to duplicate the type definitions in the IDL). This feature proves very convenient when decomposing legacy monolithic applications for use in NSI.

2. NSI components with no implementation can be useful for defining configuration information.

The fact that each component defines the interfaces it provides and uses has three key advantages:

- If a component's interface is updated, but an attempt subsequently is made to use the updated version of the component with an older version of another component, the resulting type mismatch can be diagnosed quickly and easily. Forcing the programmer to change explicitly the interface definition encourages him to think about how the implementation of the affected component needs to change.
- A component may declare only a subset of interfaces it uses or provides. This means that a component does not need to implement all the functions of an exported interface and only needs to declare the functions and attributes of an imported interface. This results in a form of interface polymorphism that gives many of the advantages of classical interface inheritance in Object-Oriented Programming (OOP), but without inheritance's associated disadvantages [9][10], and without requiring the programmer to embrace OOP.
- Declaring the interfaces separately for each component that uses them avoids needing centrally held interface definition.

Interface evolution is well supported because it is trivial to export new functions from an interface (i.e., extend an interface).

2.2 Hello, world!

To give you a taste of NSI interfaces, we present here the interface definition and associated implementation for a simple 'Hello world' NSI component. Below is the dependency file for our component.

```
#include <nsi/types.h>
#include <nsi/version.h>

supervisor module {

   /* Export component meta-data and a constructor */
   interface component {
      /* Component info attributes */
      const nsi_uuid_t iid = "204f0640-5c25-4f80-8f3e-08c0ef562ba2";
      const nsi_uuid_t cid = "476bedfc-3ad0-4a5f-8532-a3206d88e489";
      const nsi_version_t build  = MAKE_VERSION (1, 0, 0);
      const nsi_string_t name    = "helloworld";
     const nsi_string_t descr   = "Displays `hello world' string on stdout";

   };

   /* Export the application interface so we can be started */
   interface application {
      const nsi_uuid_t iid = "d185e308-eb00-11d4-8bb2-000103311257";
      const nsi_string_t name = "hello1";
```

```
 nsi_result_t Start (nsi_int32_t argc, nsi_string_t args);
   };

   /* Import the Write function of stdfile interface */
   foreign loadtime required interface stdfile {
      const nsi_uuid_t iid == "4657dcd2-eb00-11d4-8a8c-000103311257";

      ssize_t Write (int fd, const void *buf, size_t sz);
   };
};
```

Like all NSI components, our 'Hello world' component exports the `component` interface. The `component` interface contains meta-data describing the component. It can also contain entry-points for calls by the NSI runtime (see Section 3), but these functions are not implemented by this simple component. Most components export at least one other interface (to do useful work)—the `application` interface in this example. Exporting this interface means that our component can be invoked as an application (for example, a shell component might import the application interface to allow invocation of our helloworld component from a command-line).

Our sample component also imports one interface: `stdfile`. This gives POSIX-style file-access. Note that our component only imports one function from the stdfile interface: `Write`. There are many other functions defined on this interface (`Open`, `Close`, `Read`, `MkDir`, etc.); components only need explicitly import those functions that they use.

The imported interface is denoted by use of the keyword `foreign`. There are two other import modifiers here: `loadtime` and `required`. The former dictates that the import link shall be created automatically before the start of the component's lifetime (see Section 3.2); the latter that the system guarantees the availability of a provider (see Section 3.3).

Note that an interface's unique name is encoded in the Dep IDL as though it were an attribute called `iid`. This attribute is required for all interfaces. In fact, the interface's name is equivalent a mandatory data attribute; it is encoded separately in the Dep IDL to keep the syntax closer to other IDLs.

The implementation of our component in C is very simple. This is shown below:

```
#include "hello.dep.h"

nsi_result_t Start (nsi_int32_t argc, nsi_string_t args) {

   static const char msg[] = "Hello world\r\n";

   (void) _nsi_view.stdfile->Write (1, msg, (sizeof msg) - 1);

   return 0;
}
```

Our 'Hello world' component exports two interfaces, but only one function. Therefore, the implementation must provide the exported function. The `Start` function can be compared to the main function in C. Note that from `Start`, the component calls its imported function on the `stdfile` interface (there is only one imported interface in this trivial example).

3 The NSI runtime

As well as the mechanisms and tools used to create binary components, NSI consists of a runtime environment that host components.

3.1 Run-time imports

Because NSI components have compiled into them a description of the interfaces that they import and export, it is possible, at system build time, to check that all required imports are met, and create the loadtime import links (see Section 4).

A component can also import interfaces dynamically, at runtime, by calling on a component known as the *Dependency Manager*. (Or more precisely, by importing the services of the depmgr interface, which currently is exported only by the Dependency Manager component.)

Run-time imports are used to request an imported interface when the import criteria are not known until runtime. They can also be used to keep to a minimum the number of active components (see Section 3.2), which can be useful in resource-constrained devices.

3.2 Component life-cycles

NSI guarantees that a component will only have its exported functions called during its lifetime. As mentioned in Section 2.2, each NSI component exports the component interface. There are four functions in this interface: `Init`, `Destroy` (both described in this section), `Register` and `Unregister` (described in Section 3.3.).

`Init` can be seen as a component's constructor, and `Destroy` its destructor. If a component exports an `Init` function, it is called at the beginning of the component's lifetime. `Destroy` is called at the end of a component's lifetime. NSI guarantees that none of a component's exported functions shall be called by other components until after the component's Init function completes successfully. A component's `Destroy` function shall not be called until the component's exported interfaces are no longer imported by any other component.)

`Init` and `Destroy` do not need to be reentrant.

A component's lifetime begins when its `Init` function successfully returns and ends when the component's `Destroy` function returns (or when `Init` returns an error code). If a component does not export `Init` or `Destroy`, the duration of its lifetime is implicit. At the start of a component's lifetime, its global data must have been initialized. A component may have several lifetimes within a running system; these lifetimes will be surrounded by calls to `Init` and `Destroy`.

3.3 Import links

As shown in Section 2.1, an NSI component statically defines in its Dep file all the interfaces that it exports and imports. A component may only call functions or access attributes of one its imported interfaces when an import link exists with a possible exporter. There are currently two kinds of imports that a component may make: loadtime and runtime. A component's loadtime import link is created automatically before the start of the importer's lifetime. The creation of a runtime import link is initiated by the importer's implementation during its lifetime.

Interface imports are either optional or required. NSI guarantees that an import link will be created for each required loadtime import before the start of it's the importer's lifetime.

NSI guarantees that a component will be active at least as long as it has importers.

Upon the import link creation, NSI calls the exporter's `Register` function. By returning an error code from its `Register` function, an exporter may reject an importer and refuse an import link. When an import link is destroyed, NSI calls the exporter's `Unregister` function. The `Register` function is typically used by an exporter to allocate and initialize importer specific data.

Note that since the majority of imports in NSI systems are loadtime, the beginning of a component's lifetime can result in dozens of components becoming active due to recursive resolution of loadtime dependencies.

3.4 Interface stubs

Interface stubs are known in some other systems as interceptors, or before/after/around aspects [11]. A stub is a small piece of code, injected at runtime between the call-site of a importer and the entry point of the exporter. The stubs are constructed at runtime to maximize efficiency. Off-line stub generation means that (a) stubs must be created in all cases "just in case" they're needed, and (b) that the stubs must be generic, which effects how tightly optimized they can be [12].

Stubs are currently used to provide transparent distributed communication between NSI components residing on different machines connected by a network (known as DNSI).

It is also anticipated that stubs will provide a useful and practical way to provide many services in the future, from logging inter-component invocations, to automatic serialization of function calls to providing security checks.

4 Building systems from NSI components

As well as the depc interface compiler tool to generate components (see Section 2), NSI is supplied with various other tools. The most notable of these is NexBuilder.

NexBuilder is a graphical tool used to create systems from a repository of NSI components. Because each NSI component is compiled with a static description of the interfaces it imports, NexBuilder is able to determine that all required imports in a given system are met. In addition, NexBuilder can recursively satisfy various inter-component dependencies to quickly and conveniently construct a system. That is, the user of the NexBuilder tool simply drags-and-drops a few "base components", and in response NexBuilder will recursively add components to the selection as necessary to satisfy component dependencies. In the cases where there is more than one possible match for dependency, NexBuilder interacts with the user to select a component.

The input to NexBuilder is a set of components and user choices, and its output is an executable image for a given architecture and OS. The executable contains all the selected components, linked together with the NSI runtime.

5 Conclusion

NSI is a simple yet powerful component model, suited to the componentization of embedded software, particularly consumer electronics.

We believe that to succeed in the CE industry, a component-model must be both simple and flexible. Simple so that it may be used in a variety of devices, and so that it may be understood and practically employed by software engineers. A large part of NSI's advantages lies in its simplicity. Unlike many other component models, it does not incorporate an object model, or advanced features like exception handling. While this "lack of features" may represent a drawback in other arenas, for embedded applications the lack of baggage is very useful. It means that programmers have less to learn, and the use of the component model impacts minimally on the hardware resources necessary to execute NSI programs.

Perhaps most importantly for its pragmatic use in industry, NSI's simplicity makes it practical for use in decomposition of existing, monolithic or modular software. That is, the CE industry has a large amount of legacy code in existence, most of which is written in C and assembler. NSI allows this code to be decomposed with relative ease.

NSI interfaces are simple yet powerful, and offer good support for features such as interface polymorphism and interface evolution in a non-OO (object-oriented) context.

NSI is a good solution for CE vendors that want to adopt CBSE. Using NSI for code reuse has shown particular promises. Although effective software reuse depends on more than tools: development processes and cultures must also be adapted to effectively achieve code reuse [16]. NSI can be of significant assistance in such efforts.

References

[1] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass., 2000.

[2] Sara Williams and Charlie Kindel. The component object model: A technical overview. http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp, Oct 1994.

[3] Microsoft Corporation. .NET homepage. http://microsoft.com/net/.

[4] NexWave Solutions. Component-Based Software-Engineering for Consumer Electronics. http://www.nexwave-solutions.com/papers/cbse.pdf, 2004.

[5] Clemens Szysperski. Component Software: Beyond Object-Oriented Programming. Addison Wesley, 2002.

[6] The Object Management Group. CORBA Components. http://www.omg.org/docs/formal/02-06-65.pdf, June 2002.

[7] The Object Management Group. Light Weight CORBA Component Model (revised submission). http://www.omg.org/docs/realtime/03-05-05.pdf, 2004.

[8] The Open Group. OSF DCE Application Development Guide v. 3. Prentice Hall, August 1995.

[9] Ivan Ryant. Why Inheritance Means Extra Trouble. Communications of the ACM, 40(10):118-119, October 1997.

[10] John Hunt, Alex McManus, Fred Lond, and Edel Sheratt. Inheritance considered harmful. http://www.jaydeetechnology.co.uk/planetjava/tutorials/design/InheritanceConsideredHarmful.PDF.

[11] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220-242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[12] Eugen-Nicolae Volanschi, Gilles Muller, and Charles Consel. Safe Operating System Specialization: the RPC Case Study. In 1st Workshop on Compiler Support for System Software, February 1996.

[13] Oscar Nierstrasz. Putting Change at the Center of the Software Process. In 7th International Symposium, CBSE, May 2004.

[14] Manny M. Lehman and Les Belady. Program Evolution -- Processes of Software Change. London Academic Press, 1985.

[15] Richard Riehle. Ada Distilled: An Introduction to Ada Programming for Experienced Computer Programmers. http://www.adapower.com/learn/adadistilled.pdf, 2003.

[16] Maurizio Morisio, Michel Ezran, and Colin Tully. Success and failure factors in software reuse. IEEE transactions on computer systems, 28(4), April 2002.