To appear in the proceedings of the International Conference on Generative Programming and Component Engineering (GPCE'06)

Creating Custom Containers with Generative Techniques

Gabriel A. Moreno

Software Engineering Institute, Carnegie Mellon University

gmoreno@sei.cmu.edu

Abstract

Component containers are a key part of mainstream component technologies, and play an important role in separating non-functional concerns from the core component logic. This paper addresses two different aspects of containers. First, it shows how generative programming techniques, using AspectC++ and meta-programming, can be used to generate stubs and skeletons without the need for special compilers or interface description languages. Second, the paper describes an approach to create custom containers by composing different non-functional features. Unlike component technologies such as EJB, which only support a predefined set of container types, this approach allows different combinations of non-functional features to be composed in a container to meet the application needs.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Design

Keywords Aspect-oriented programming, AspectC++, component, container, generative programming, meta-programming, non-functional concern.

1. INTRODUCTION

Component containers are a key part of mainstream component technologies such as Enterprise JavaBeans (EJB), and CORBA Component Model (CCM). Containers mediate the interaction of a component with the runtime environment and with other components. Furthermore, they relieve the developer from dealing with routine but nevertheless error-prone tasks such as allocating resources, creating threads, performing inter-process communication (IPC), etc. In that way, the component developer can focus on the functional logic of the component and rely on the container to perform those tedious tasks, either by letting the container carry them out implicitly or by explicitly calling services the container provides. The concept of containers provides so much leverage to component-based software development that it has been adopted even in lightweight component models [5, 9].

In addition to being advantageous from the software construction perspective, container-based component technologies provide the foundation for supporting analysis and predictability. For example, the Cadena[1] project at Kansas State University [8] and the PACC[2] initiative at the Carnegie Mellon Software Engineering Institute (SEI) [20] are both creating methods to analyze and predict the behavior of systems developed with container component technologies, so that components can be used in real-time and safety critical systems.

This paper addresses two different facets of containers. First, it shows how generative programming techniques [3] can be used to generate containers with typed stubs and skeletons (i.e., the client- and server-side meta-objects [22]) so that programmers are relieved from writing marshaling and unmarshaling code, and type checking can be done statically. Stubs and skeletons are usually generated by a compiler that either takes a description of the component's interface in an interface description language (IDL) or uses reflection to get the component's interface. For example, EJB containers are generated by special compilers that in turn use the RMI compiler to generate the stubs and skeletons. The approach discussed in this paper generates stubs and skeletons without the need for special compilers and a complex infrastructure by combining aspect-oriented programming (AOP) using AspectC++ [16] with template meta-programming [19], a powerful combination already demonstrated by Lohmann and colleagues [13].

The other facet of containers this paper attends to is about the features, services, and/or policies implemented by the container. These are referred to as *non-functional aspects/services/properties/concerns* in the literature [1, 2, 4] because they are not directly related with the function that a component has to carry out. Instead, they often provide mechanisms to guarantee quality attributes such as performance, security, availability, etc. In general, containers implement a fixed set of non-functional concerns. In EJB for instance, a different container is generated depending on the kind of EJB (session, entity, or message-driven), but there is no provision for customizing the container by composing different non-functional concerns. This paper demonstrates how non-functional features can be composed to create custom containers that fit the application's needs. Thus, illustrating how a lightweight component technology was enhanced with support for compositional adaptation [14] by using AspectC++ as a composition mechanism.

The ideas in this paper were implemented using Pin [9], a lightweight component technology. One of the fundamental characteristics of Pin is that it implements the container idiom; the functional logic of the component—the component core—is encapsulated by a container that implements coordination and scheduling policies, connectors, and other services. The container enforces the constraints and assumptions of quality attribute theories [10], and prevents unexpected component interactions, thus leading to predictable behavior.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-237-2/06/0010.

[1] Component Architecture Development ENvironment for Avionics systems.

[2] Predictable Assembly from Certifiable Components

The rest of the paper is organized as follows. Section 2 gives an overview of the Pin component model. Section 3 explains the generative approach to create stubs and skeletons. Section 4 shows the aspect-oriented implementation of non-functional concerns. An example of a non-functional feature implementing the sporadic server algorithm [12] is presented. Section 5 refers to related work and Section 6 concludes the paper.

2. PIN COMPONENT MODEL OVERVIEW

Pin is a simple component technology, suitable for embedded systems, that supports pure assembly [9]. Components are assembled together by connecting a *source* pin to a *sink* pin without the need for "glue" code. A source pin produces a stimulus, which is then received by a sink pin, causing the component to react to the stimulus. The behavior of a Pin component is encapsulated in a *reaction*, which, in addition to performing computations, can in turn emit other stimuli through its source pins.

Pin allows both synchronous and asynchronous interactions. The former has the traditional call/return semantics, while the latter has event-based semantics. In both kinds of interactions, the stimuli are messages that may carry data along with them. For that reason, pins have a data interface or signature that specifies the parameters that they produce and consume. For a connection between a source and a sink to be valid, the signature of the sink pin must be the complement of the signature of the source, because the parameters produced by the source must be consumed by the sink and vice versa.

An important characteristic of Pin is that it realizes the container idiom. The pre-fabricated container wraps the custom code, mediating all its interactions with the environment and other components. The container implements the message loop that waits for incoming messages on sink pins, handles timeouts, and dispatches the corresponding reaction handler function in the appropriate thread upon receiving a message. In addition, it provides services to reply to synchronous requests, and to make other synchronous and asynchronous requests through the source pins. Furthermore, all these services make the location of other components transparent since the container handles IPC across threads, processes, and networked computers.

Besides providing all these services, different containers may implement different features or policies. For example, a container may implement a specific scheduling policy. In addition, Pin supports the dynamic binding of containers and custom code (also referred to as *component core*) so that the same component core can be used with different containers to fit the application needs. Figure 1 depicts the mechanism provided by Pin to support the dynamic binding of containers and custom code to create a component[3] [10]. The container uses the *ComponentCore* interface both to get information about the component and to dispatch different handlers at different points in the component life cycle. The custom code in turn uses the *ContainerServices* interface to access the services provided by the container. Tables 1 and 2 show the members of these interfaces.[4]

This brief description of Pin is limited to the concepts relevant to this paper. More details and the rationale behind Pin can be found in the work of Hissam and colleagues [9].

[3] Although an instantiable Pin component is created by loading a component core into a container, throughout this paper the terms *component* and *component core* are used interchangeably.

[4] Only the members relevant to this paper are shown.

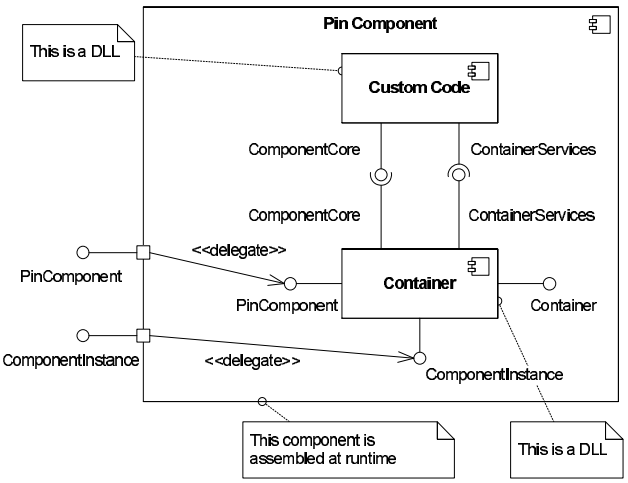

Figure 1. Pin Component and Container

Table 1. ComponentCore Interface

Member	Description
reactionsInfo[]	Description of the reactions in a component, including a pointer to the reaction handler function
createComponentInstance()	Initializes the internal state of a component instance
deleteComponentInstance()	Deletes the internal state of an instance being deleted
reactionInitialize()	Performs operations needed upon initializing a reaction of a component instance
reactionTerminating()	Performs operations required upon terminating a component instance reaction

Table 2. ContainerServices Interface

Member	Description
sendOutSourcePin()	Sends an asynchronous message through a source pin
sendOutSourcePinWait()	Sends a synchronous message through a source pin
sendReply()	Sends a reply to a received synchronous message
parseUserMessage()	Parses a PIN_MSG received by a reaction handler

3. GENERATION OF STUBS AND SKELETONS: THE INNER CONTAINER

A stub is a client-side meta-object that is a proxy for a remote component [22]. It allows the caller of a component to invoke it by means of a simple procedure call. Behind the scenes, the stub marshals the arguments, sends the request to the component being called, waits for the reply, and unmarshals the returned values. The counterpart of the stub is the skeleton, a server-side meta-object. The skeleton receives remote invocations, unmarshals the arguments, invokes the component being called, marshals the re-

sults, and finally sends them back to the caller. The skeleton makes the implementation of a function in a component as easy as writing a regular function or method.

3.1 Motivation

A Pin container provides a large part of the mechanism required for component interactions. Nevertheless, the component developer still needs to take care of marshaling and unmarshaling arguments and return values, and invoking the appropriate container services to send messages and replies to other components.

In order to illustrate what the component code looks like, a simple assembly is shown in Figure 2. The *SensorMonitor* component has one asynchronous sink pin (*trigger*) that is stimulated periodically by a clock. This component reads a sensor and sends a synchronous request through a source pin to the *Converter* component to get the sensed values converted to physical units. If the result surpasses a threshold, *SensorMonitor* sends an alarm message through an asynchronous source pin to the *ConsoleOutput* service. Listing 1 shows the source code for the reaction handler of *SensorMonitor*. Line 3 checks whether the message corresponds to a pin message. In this component, other types of messages such as timeouts are ignored. Also, in this reaction handler, there is no need to determine through which sink pin the message arrived since the component has only one sink pin. Lines 6–7 are related to the functional logic of the component; they read the two sensors needed to compute the physical magnitude. The next step in the functional logic is to send a synchronous request through a source pin to convert the values obtained from the sensors to physical units. The signature of that pin can be specified using CCL, a language used to describe interfaces (as in an IDL), to specify the behavior of reactions, and to define assemblies of components [21]. The signature of the *convert* source pin in CCL would be

```
source synch convert(produce int a, produce int b,
                     consume float result);
```

meaning that it is a synchronous source pin named *convert*, that sends two values of type *int* and expects a value of type *float* as a reply from the synchronous interaction. Lines 9–26 make the synchronous interaction through the *convert* source pin. Lines 10–12 define some needed variables, lines 15–20 marshal the arguments, and lines 21–23 invoke the container service to send the synchronous message and wait for the reply. Line 26 unmarshals the result of the synchronous request. Line 28 checks if the result surpassed the threshold—a functional logic step—and if that is the case, lines 29–34 marshal an alarm message and send it out through an asynchronous source pin.

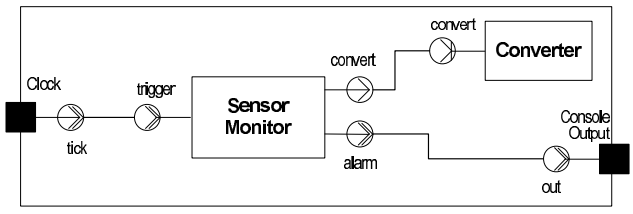

Figure 2. Running Example Assembly

Listing 2 shows the code for the reaction associated with the *convert* sink pin in the *Converter* component. Although the Pin container dispatches this code through a function call, the component developer still has to deal with several things besides the functional logic. Several data structures are used to receive parameters and send results (lines 3–6); the reaction handler can be invoked for events of other kinds (such as timeouts) that do not correspond to sink pin activation (lines 8–14); likewise, the same reaction handler

```
1  BOOL reactionHandler(Reaction* pReaction,
2                       CommonMessage *pMsg) {
3      if (pMsg->type != PIN_MSG) {
4          return TRUE; // don't care about others
5      }
6      int readingA = readSensor(0);
7      int readingB = readSensor(1);
8
9      /* request conversion */
10     IpcPort_Message message;
11     IpcPort_Message answer;
12     int answerDataSize;
13
14     /* marshal arguments */
15     int* pNextArg = (int*) message.data;
16     int dataSize = 0;
17     *pNextArg++ = readingA;
18     dataSize += sizeof(int);
19     *pNextArg++ = readingB;
20     dataSize += sizeof(int);
21     sendOutSourcePinWait(pReaction, SOURCE_CONVERT,
22           &message, dataSize, &answer,
23           &answerDataSize, IPCPORT_WAITFOREVER, 0);
24
25     /* unmarshal results */
26     float result = *((float*) answer.data);
27
28     if (result > THRESHOLD) { // send alarm
29         SPrintf((char*) message.data,
30                 "ALARM: SensorMonitor\n");
31         sendOutSourcePin(pReaction, SOURCE_ALARM,
32               &message,
33               strlen((char*) message.data) + 1,
34               IPCPORT_WAITFOREVER);
35     }
36     return TRUE;
37 }
```

Listing 1. Original SensorMonitor Reaction Handler Code

is invoked for all the sink pins in that reaction (lines 16–17); the parameters consumed by the sink pin need to be unmarshaled (lines 18–23); and finally, the results have to be marshaled and sent back by calling a container service (lines 34–38). Only lines 25–32 are directly related to the functional logic of this component.

Even though the marshaling/unmarshaling method used in the example is very basic, most of the code in the handler functions is not directly related to the functional logic of the components. Moreover, the code specific to the logic is entangled with the non-functional code. In order to avoid these issues and let the developer focus on the functional logic, it is desirable to automatically generate the stubs and skeletons for the source and sink pins respectively.

Due to their role, stubs and skeletons belong to the container. However, they are specific for a particular component. If they were made part of a Pin container, the feature of Pin that allows a developer to exchange containers and components would be broken. Consequently, it makes sense to keep the containers component-independent, and put the stubs and skeletons together in an *inner container*, a container that not only is tailored for a particular component, but also fits inside a regular Pin container. Given that the inner container is custom-made for a particular component, when compiled it becomes part of the component core binary.

3.2 Generative Approach to Stub and Skeleton Generation

Mainstream component technologies such as EJB and CORBA provide an infrastructure to generate the stubs and skeletons. In the case of CORBA, the interface of a component has to be specified in an IDL, which is then processed by a special compiler to

```
1  BOOL reactionHandler(Reaction* pReaction,
2                       CommonMessage* pMsg) {
3     IpcPort_Message* pMessage;
4     IpcPort_MessageInfo messageInfo;
5     IpcPort_Message messageOut;
6     int dataSize;
7
8     if (pMsg->type != PIN_MSG) {
9        return TRUE; // don't care about others
10    }
11
12    // get pin message
13    pMessage = parseUserMessage(pReaction, pMsg,
14                       &dataSize, &messageInfo);
15
16    switch (pMessage->sinkPin) {
17       case SINK_CONVERT:
18          /* unmarshal arguments */
19          unsigned char* pNextArg
20                             = pMessage->data;
21          int sampleA = *((int*) pNextArg);
22          pNextArg += sizeof(int);
23          int sampleB = *((int*) pNextArg);
24
25          /* functional logic */
26          float physicalA
27             = (sampleA - BASELINE_A)
28                   * ADUS_PER_PU_A;
29          float physicalB
30             = (sampleB - BASELINE_B)
31                   * ADUS_PER_PU_B;
32          float result = physicalA / physicalB;
33
34          /* marshal result and send reply */
35          *((float*) messageOut.data) = result;
36          sendReply(pReaction,
37                    &messageInfo, &messageOut,
38                    sizeof(float));
39          break;
40       default:
41          return FALSE; // wrong sink pin
42    }
43    return TRUE;
44 }
```

Listing 2. Original Converter Reaction Handler Code

generate the stub and skeleton. EJB takes advantage of the reflection capabilities of Java to determine the methods in a component and their signatures. Although this overcomes the need for using an IDL, a special compiler called *ejbc*[5] is still used to generate the stub and skeleton. Instead of using a special compiler, the approach described here exploits the generative capabilities of template metaprogramming [19] to generate the stub and skeleton.

Templates make C++ a two-level language, where the static code, in the form of template meta-programs, is evaluated at compile-time, and the dynamic code is executed at runtime [3].

In addition to being able to generate code, a reflection mechanism is needed to determine the signature of the pins so that the corresponding stub/skeleton code can be generated. Moreover, in order to generate the code at compile-time, compile-time reflection is needed. AspectC++ has a compile-time join point API that can be used to determine the number of arguments in a join point and their type [16], thus providing the needed reflection mechanism.

[5] Some EJB implementations relieve the developer from explicitly invoking the compiler to create the meta-objects, because that process is implicitly carried out when the component is deployed.

3.2.1 Source Pin Stubs

The objective of a source pin stub is to make the interaction through a source pin as simple as making a function call. Therefore, a function that represents the stub for the source pin is declared. For the *convert* source pin, the declaration is

```
int source_synch_0(Reaction* pReaction,
                   int a, int b, float* result);
```

This is very similar to the CCL signature of the pin. Instead of using the name of the pin, the index of the pin (0 in this case since it is the first source pin) is used in the name of the function. The parameter *pReaction* is akin to the implicit *this* parameter that member functions in C++ have. In this case, it has to be passed explicitely because Pin was implemented in a way such that C language could be used to develop components. Next, the parameters of the pin follow. Considering that pins can produce and consume multiple parameters, a convention is used to distinguish them: produced parameters are declared as passed by value, whereas consumed parameters are declared as passed by pointer. In addition, the stub function has a return value of type *int* so that success or failure can be reported back to the caller.

All that is required from the component developer is declaring—but not implementing—the function that represents the source pin. The implementation of that stub function is generated automatically. The generation is done as follows. First, it has to be determined where the generated code should be inserted or woven in. In the aspect shown in Listing 3, an *around* advice is defined in line 2 for the pointcut matching calls to the stub functions. Since the aspect advises the call (as opposed to the execution) and *proceed*() is never called for that join point, there is no need to ever define the stub function that the component developer declared. The calls to the stub are replaced with the code in the advice.

The first step the stub needs to take is to marshal the arguments, and this is done by code that is automatically generated specifically for that stub by a template meta-program. Only the produced parameters (a and b in this example) need to be marshaled. However, for the sake of simplicity, let us assume for a moment that all the parameters, both consumed and produced, have to be marshaled. Listing 4 contains a basic implementation of the marshaling meta-program. The approach demonstrated by Spinczyk and colleagues [17] to iterate with a meta-program over the list of arguments in a join point is used. The meta-function *MarshalParams* in lines 11–16 takes two arguments.[6] The first one is the *JoinPoint* class corresponding to the join point of the advice. The second argument is an integer used to control the iteration over the arguments. The code this meta-function generates is a call to the *marshal* function (line 13) with the appropriate argument type, which is known thanks to the *TJP* argument of the template. Line 14 recursively uses the same meta-function to generate the code for the next parameter. To end the recursion, the template is specialized for N=1 (lines 18–22). The generated code can be executed by the dynamic code within the advice by calling the *execute* method as follows:

```
int dataSize = MarshalParams<JoinPoint,
                             JoinPoint::ARGS - 1>
               ::execute(&message, 0, tjp);
```

The parameters to this method are an *IpcPort_Message* where the parameters must be marshaled; the *offset* of the next parameter to be marshaled; and the particular instance of the *JoinPoint* class, which allows accessing the actual value of the parameters. The return value of this method is the offset of the next parameter, which at the end of the recursion is the size of all the marshaled parameters. Since the C++ compiler can optimize the code, the generated code will not be recursive at all. In fact, because all the functions are

[6] Note that the complete version of *MarshalParams* takes three arguments. However, the third argument is not needed in the this simpler version.

```
aspect SourcePinStub {
    advice call("int source_synch_%(Reaction*, ...)") : around () {
        IpcPort_Message message;
        IpcPort_Message answer;
        int answerDataSize;
        Reaction* pReaction = *tjp->arg<0>();

        short dataSize = MarshalParams<JoinPoint, JoinPoint::ARGS - 1, SourceProduce>
                         ::execute(&message, 0, tjp);

        int pinId = JoinPoint::signature()[17] - '0'; // yes, only 1 digit index
        if (sendOutSourcePinWait(pReaction, pinId, &message, dataSize,
                                 &answer, &answerDataSize, IPCPORT_WAITFOREVER, 0)) {
            UnmarshalParams<JoinPoint, JoinPoint::ARGS - 1, SourceConsume>::execute(&answer, 0, tjp);
            *tjp->result() = TRUE;
        } else {
            notifyController(pReaction->pInstance, CONTROLLER_UNKNOWN_ERROR,
                             "sendOutSourcePinWait failed");
            *tjp->result() = FALSE;
        }
    }
};
```

Listing 3. Aspect for Source Pin Stub

```
inline int marshal(IpcPort_Message* message, int offset, int& value) {
    *((int*) (message->data + offset)) = value;
    return offset + sizeof(int);
}

inline int marshal(IpcPort_Message* message, int offset, float& value) {
    *((float*) (message->data + offset)) = value;
    return offset + sizeof(float);
}

template<class TJP, int N> struct MarshalParams {
    static inline int execute(IpcPort_Message* pMessage, int offset, TJP* tjp) {
        int newOffset = marshal(pMessage, offset, *tjp->template arg<TJP::ARGS - N>());
        return MarshalParams<TJP, N - 1>::execute(pMessage, newOffset, tjp);
    }
};

template<class TJP> struct MarshalParams<TJP, 1> {
    static inline int execute(IpcPort_Message* pMessage, int offset, TJP* tjp) {
        return marshal(pMessage, offset, *tjp->template arg<TJP::ARGS - 1>());
    }
};
```

Listing 4. Simple Marshaling Meta-program

declared as *inline*, the generated code will not have any subroutine calls after the optimization.

An important benefit of this approach is that type checking is done at compile time. The *marshal* function is overloaded for the different supported parameter types. However, if a stub for a source pin is declared with a parameter type for which a marshal function has not been implemented, the compiler will throw an error. Note that in the example, the *value* parameter of the *marshal* functions has been declared as a reference to avoid implicit type conversions, which would go undetected and cause problems.

Now, the fact that only the produced parameters have to be marshaled has to be dealt with. The convention of using pointer types to distinguish consumed from produced parameters was used. However, there is no direct way to determine whether a type passed as a template argument is a pointer or not. To solve this, trait templates [3][15], a method to represent meta-information of types, is used. For a given type, it must be determined whether parameters of that type must be included in a marshaling/unmarshaling operation, and whether that type is a pointer type or not. Listing 5 shows how this information is encoded in trait templates. Lines 1–4 define a generic trait template with defaults for what is not a source produced parameter. Lines 6–12 specialize the trait template overriding the traits for the types that represent parameters produced by a source pin (i.e., *int* and *float* for the running example). The following examples, both of which evaluate to *true*, show how the trait template is used.

```
SourceProduce<int>::include == true;
SourceProduce<int*>::include == false;
```

Lines 14–19 define a trait template for the types representing parameters consumed by a source pin by simply including all the types that are not included as produced.

The meta-information encoded in the traits is used to select alternative pieces of code when the code is being generated. That can be done with the *IF* meta-function [3], which is used as follows:

```
1  template<class T> struct SourceProduce {
2      static const bool include = false;
3      static const bool isPointer = true;
4  };
5
6  template<> struct SourceProduce<int> {
7      static const bool include = true;
8      static const bool isPointer = false;
9  };
10
11 template<> struct SourceProduce<float>
12             : public SourceProduce<int> {};
13
14 template<class T> struct SourceConsume {
15     static const bool include
16             = !SourceProduce<T>::include;
17     static const bool isPointer
18             = SourceProduce<T>::isPointer;
19 };
```

Listing 5. Trait Templates

IF<condition, ThenClass, ElseClass>::RET object;

If the condition is true, *object* will be an instance of *ThenClass*, otherwise, it will be an instance of *ElseClass*. Listing 6 shows the code of the parameter marshaling meta-program with correct handling of consumed and produced parameters. The first change with respect to the simplified version is that the *MarshalParams* template takes now a third parameter *ArgMetaInfo*, which is another template itself, namely, the traits template. In line 20, a helper type is defined by instantiating *ArgMetaInfo* with the type of the parameter being considered for marshaling. Instead of generating a call to the *marshal* function directly as it was done before, a code selection technique is used to instantiate the template with the appropriate code for marshaling a parameter passed by value, marshaling a parameter passed by pointer, or skipping a parameter if it does not have to be included in the marshaling operation (lines 23–27). For instance, if the parameter has to be marshaled (i.e., *argMetaInfo::include* is true) and the parameter is not a pointer, the template *MarshalArgValue* is instantiated to generate the code that will be executed by calling *execute()*.

Listing 7 shows the source code of the reaction handler for the *SensorMonitor* component using the automatically generated stubs. Comparing it to the original code in Listing 1, it can be appreciated that it is more compact and that the functional logic of the component is easier to follow as it is not entangled with the marshaling/unmarshaling code.

3.2.2 Sink Pin Skeletons

The skeleton is the counterpart of the stub. Its purpose is to make the invocation of the reaction code associated with a sink pin to look as if it had been called by a regular procedure call without additional overhead to the programmer.

With this approach of generated skeletons, the component developer defines a function with the signature of the sink pin and a special first argument that is explained later. This function, shown in lines 1–10 of Listing 8, only needs to implement the functional logic of the sink pin. Except for the first argument, the signature of the function matches the signature of the corresponding sink pin, which in CCL is declared as:

sink mutex convert(**consume int** a,
consume int b, **produce float** result);

Note that this signature is the complement of the one for the source pin in the *SensorMonitor* component. However, in the C++ version of the signature, the interpretation of parameter types is reversed;

```
1  int source_synch_0(Reaction* pReaction, int a,
2                     int b, float* result);
3  int source_unicast_1(Reaction* pReaction,
4                       char* message);
5
6  BOOL reactionHandler(Reaction* pReaction,
7                       CommonMessage *pMsg) {
8      if (pMsg->type == PIN_MSG) {
9          int readingA = readSensor(pReaction, 0);
10         int readingB = readSensor(pReaction, 1);
11
12         /* request conversion */
13         float result;
14         source_synch_0(pReaction, readingA,
15                        readingB, &result);
16
17         if (result > THRESHOLD) { // send alarm
18             source_unicast_1(pReaction,
19                     "ALARM: SensorMonitor\n");
20         }
21     }
22     return TRUE;
23 }
```

Listing 7. New SensorMonitor Reaction Handler Code

```
1  void sink_mutex_convert(
2                 PinMessageData* pMessageData,
3                 int sampleA, int sampleB,
4                 float* pResult) {
5     float physicalA = (sampleA - BASELINE_A)
6                                * ADUS_PER_PU_A;
7     float physicalB = (sampleB - BASELINE_B)
8                                * ADUS_PER_PU_B;
9     *pResult = physicalA / physicalB;
10 }
11
12 BOOL pinMessageHandler(
13                PinMessageData* pMessageData) {
14    switch (pMessageData->pMessage->sinkPin) {
15       case SINK_CONVERT:
16          float result;
17          sink_mutex_convert(pMessageData,
18                             0, 0, &result);
19          break;
20       default: return FALSE;
21    }
22    return TRUE;
23 }
```

Listing 8. New Converter Reaction Handler Code

parameters passed by value are consumed in the sink pin, whereas those passed by pointer are produced.

Unfortunately, the solution to insert the skeleton code is not as neat as for the stubs. The reason is due to the use of the join point API of AspectC++ as a reflection mechanism, which can only detect the existence of functions that are called from some place in the code. Therefore, the component developer has to write a function named *pinMessageHandler* that includes a call to the function corresponding to a given sink pin (see lines 12–23 in Listing 8). The call to the sink pin function in line 17 has two special requirements. First, it has to forward the *PinMessageData* structure that *pinMessageHandler* received; second, it must pass valid addresses for the produced parameters passed by pointers. Note, however, that for the consumed parameters passed by value,

```
template<class TJP, int argIndex> struct MarshalArgPointer {
    static inline int execute(IpcPort_Message* pMessage, int offset, TJP* tjp) {
        return marshal(pMessage, offset, **tjp->template arg<argIndex>());
    }
};

template<class TJP, int argIndex> struct MarshalArgValue {
    static inline int execute(IpcPort_Message* pMessage, int offset, TJP* tjp) {
        return marshal(pMessage, offset, *tjp->template arg<argIndex>());
    }
};

template<class TJP> struct SkipArg {
    static inline int execute(IpcPort_Message* pMessage, int offset, TJP* tjp) {
        return offset;
    }
};

template<class TJP, int N, template<class T> class ArgMetaInfo> struct MarshalParams {
    typedef ArgMetaInfo<typename TJP::template Arg<(TJP::ARGS - N)>::ReferredType> argMetaInfo;
    static inline int execute(IpcPort_Message* pMessage, int offset, TJP* tjp) {
        int newOffset =
            IF<argMetaInfo::include,
               typename IF<argMetaInfo::isPointer,
                           MarshalArgPointer<TJP, TJP::ARGS - N>,
                           MarshalArgValue<TJP, TJP::ARGS - N> >::RET,
               SkipArg<TJP> >::RET::execute(pMessage, offset, tjp);
        return MarshalParams<TJP, N - 1, ArgMetaInfo>::execute(pMessage, newOffset, tjp);
    }
};

template<class TJP, template<class T> class ArgMetaInfo> struct MarshalParams<TJP, 1, ArgMetaInfo> {
    typedef ArgMetaInfo<typename TJP::template Arg<(TJP::ARGS - 1)>::ReferredType> argMetaInfo;
    static inline int execute(IpcPort_Message* pMessage, int offset, TJP* tjp) {
        return
            IF<argMetaInfo::include,
               typename IF<argMetaInfo::isPointer,
                           MarshalArgPointer<TJP, TJP::ARGS - 1>,
                           MarshalArgValue<TJP, TJP::ARGS - 1> >::RET,
               SkipArg<TJP> >::RET::execute(pMessage, offset, tjp);
    }
};
```

Listing 6. Marshaling Meta-program

bogus values are used because they are going to be replaced by the skeleton with values received from the source pin.

The skeleton consists of two parts: a non-generated reaction handler that is used as the component's reaction handler; and an aspect that advises calls to sink pin functions. The reaction handler (whose code is not shown for brevity), fills a *PinMessageData* structure with information about the message that the advice will need, and calls *pinMessageHandler* with this structure as an argument. The *around* advice, shown in Listing 9, unmarshals the parameters consumed by the sink by using the code generated by the *UnmarshalParams* meta-function, calls *proceed*() on the join point to execute the sink pin function, marshals the parameters produced using the *MarshalParams* meta-function, and finally replies to the synchronous request. Note that instead of defining new trait templates for sink parameters, the trait templates for sources are used in reverse order.

Despite requiring the component developer to define the *pinMessageHandler* function to overcome the limitations of using AspectC++ as a reflection mechanism, using the automatically generated skeletons represent a big improvement considering that the code needed to implement the *convert* sink pin was reduced roughly in half, and the functional logic is not entangled with non-functional code.

4. COMPOSING NON-FUNCTIONAL FEATURES

Separation of concerns has been for decades the motivation of many advances in software engineering. Not surprisingly, it has become a common trend to separate the non-functional and functional parts of an application by relying on the services provided by middleware and component technologies, specially considering that the non-functional features they provide are usually more difficult to implement than the core logic of a component [2]. Notwithstanding, even EJB and CCM fall short of fully enabling this separation of non-functional concerns because they only support a limited number of container types with predefined features. Containers should be adaptable to different requirements, which may require containers with different combinations of non-functional features. This leads to the need to compose features into containers [1, 4].

Since non-functional properties are cross-cutting concerns, it is natural to turn to the virtues of AOP to modularize and compose them. Although AOP certainly allows the non-functional features to be disentangled from the core logic, it has some drawbacks when advice is applied directly to the component. First, it needs the component's source code so that the aspects can be woven in.

```
1  aspect SinkPinSkeleton {
2     advice call("void sink_mutex_%(PinMessageData*, ...)") : around() {
3        PinMessageData* pMessageData = *tjp->arg<0>();
4        UnmarshalParams<JoinPoint, JoinPoint::ARGS - 1, SourceProduce>
5                       ::execute(pMessageData->pMessage, 0, tjp);
6        tjp->proceed();
7        IpcPort_Message answer;
8        int dataSize = MarshalParams<JoinPoint, JoinPoint::ARGS - 1, SourceConsume>
9                          ::execute(&answer, 0, tjp);
10       sendReply(pMessageData->pReaction, pMessageData->pMessageInfo, &answer, dataSize);
11    }
12 };
```

Listing 9. Sink Pin Skeleton Aspect

Second, aspects may have dependencies on elements of the internal implementation of the component.

In the approach presented here to achieve the composition of non-functional features, aspects are used to advise the container. This has several advantages over using aspects directly on the component to modularize non-functional concerns. First, the aspects can exploit the fact that every interaction of the component with its environment and vice versa not only goes through the container, but also does it through known and stable interfaces. For instance, this allows encrypting all the messages between components with a security aspect. Second, since the container controls the component life cycle, aspects can exploit the knowledge of the component's state (from the point of view of the runtime, not its internal state). Last, but not least, this approach does not require having access to the source code of the component. This in itself has the same advantages of the dynamic binding of containers in Pin, that is, a component does not need to be recompiled when a non-functional property is imposed on it, and the component developer is oblivious to the non-functional aspects that will be used on that component.

The following section describes an example of a non-functional feature implemented using this approach. Other features, such as encryption, can be implemented in the same way. Composing more than one feature in the same container only requires including the aspects for them in the build process of the container.

4.1 Sporadic Server

The sporadic server is a mechanism to schedule aperiodic tasks at a given priority, while limiting their impact on the schedulability of other tasks in the system [12]. The sporadic server reserves a certain amount of execution capacity that is used to service aperiodic events. When execution time is consumed from this budget, a replenishment is scheduled to occur one *replenishment period* later. If the execution budget is exhausted and an aperiodic event needs to be serviced, the aperiodic task is relegated to execute at background priority. The sporadic server gives the aperiodic task a good quality of service while retaining the predictable execution of periodic tasks even in the face of a burst of aperiodic events.

The implementation of the sporadic server is not trivial. In order to correctly implement the replenishment policy, events have to be waited for at the highest priority, and upon receiving one, immediately revert to the sporadic server or background priority level. In addition, a separate thread is needed to keep track and carry out the pending replenishments [7]. Even using the sporadic server requires discipline. The aperiodic task (or component) needs to create an instance of the sporadic server, call a function to *arm* the sporadic server every time it is going to enter a wait for an event, and must call another function to *request* execution budget to the sporadic server. Finally, it has to shut the sporadic server down.

The sporadic server is a perfect example of a non-functional feature that could be provided by the container. First, its complexity would most likely overshadow the core logic of the component. Second, when provided by the container, it can give strong performance guarantees both to the contained component and to the rest of the system, which result in predictability.

Listing 10 shows the aspect for the sporadic server feature. The implementation, in the C language, of the sporadic server algorithm itself is not shown, but it consists of a data structure *SporadicServer* that holds the state of the sporadic server and the functions *ssmgr_initialize*(), *ssmgr_arm*(), *ssmgr_request*(), and *ssmgr_shutdown*() that operate on it. Lines 2–3 define a map that is going to be used to get the sporadic server corresponding to a given reaction. Lines 5–9, define the parameters of the sporadic server. In line 11, a pure virtual method *configure*() is declared to make this aspect abstract so that a particular use of the aspect can be parameterized by aspect inheritance. Then three advices are specified corresponding to initialization, normal operation, and shutdown. Note that the pointcuts seem to refer to functions in the *ComponentCore* interface; however, they refer to proxy functions to that interface that reside in the container. In that way, the aspect can be woven in the container.

The advice in lines 13–21, is executed after the execution of *reactionInitialize*(), that is, when the thread for the reaction is about to be created and enter the message loop. The sporadic server is created, initialized, and associated to the reaction. The priority of the reaction is set to *SS_ARM_PRIORITY*, which has the same effect as calling *ssmgr_arm*() as soon as the thread is created. Lines 23–30 show the *around* advice for the execution of the reaction handler. Basically, it calls *ssmgr_request*(), to request for execution budget. Upon returning from that function, the priority will have been set to either the sporadic server level or background, as decided by the sporadic server algorithm. Then, *proceed*() is called on the join point to execute the reaction handler. Finally, *ssmgr_arm*() is called to arm the sporadic server for waiting for the next event. The advice in lines 32–37 takes care of shutting the sporadic server down when the reaction thread terminates.

Due to the fact that the feature needs to be configured with the desired parameters for the sporadic server when used, an aspect derived from *SporadicServerAspect* is defined to perform this configuration (see Listing 11).

With this approach, any Pin component core can be made compliant with the sporadic server scheduling algorithm without any modification. It just needs to be deployed in a container in which this feature has been composed.

5. RELATED WORK

Gal et al. [6] used aspects to create client- and server-side meta-objects in order to separate the component code from middleware-specific code. However, those aspects were not generic due to the lack of a reflection mechanism in AspectC++ at the time. The

```
aspect SporadicServerAspect {
   typedef std::map<Reaction*, SporadicServer*> ReactionSSMap;
   ReactionSSMap reactionSSMap;

   int replenishmentPeriod;
   int budget;
   int normalPriority;
   int backgroundPriority;
   int executionTime;

   virtual void configure() = 0;

   advice execution("void reactionInitialize(Reaction*)") && args(pReaction)
            : after(Reaction* pReaction) {
      SporadicServer* pSs = (SporadicServer*) Malloc(sizeof(SporadicServer));

      ssmgr_initialize(pSs, replenishmentPeriod, budget, normalPriority,
                       backgroundPriority);
      pReaction->priority = SS_ARM_PRIORITY;
      reactionSSMap[pReaction] = pSs;
   }

   advice execution("int reactionHandler(Reaction*, CommonMessage*)")
              && args(pReaction, pMsg)
            : around(Reaction* pReaction, CommonMessage* pMsg) {
      SporadicServer* pSs =  reactionSSMap[pReaction];
      ssmgr_request(pSs, executionTime);
      tjp->proceed();
      ssmgr_arm();
   }

   advice execution("void reactionTerminating(Reaction*)") && args(pReaction)
            : after(Reaction* pReaction) {
      SporadicServer* pSs =  reactionSSMap[pReaction];
      ssmgr_shutdown(pSs);
      reactionSSMap.erase(pReaction);
   }
};
```

Listing 10. Sporadic Server Aspect

```
aspect ThisSporadicServer
         : public SporadicServerAspect {

   ThisSporadicServer() {
      configure();
   }

   void configure() {
      replenishmentPeriod = 80;
      budget = 10;
      normalPriority = 6;
      backgroundPriority = 100;
      executionTime = 10;
   }
};
```

Listing 11. Sporadic Server Aspect Configuration

approach to generate stubs and skeletons presented in this paper uses the generic advice technique shown by Lohmann et al. [13].

Conan et al. [2] argued that non-functional services should be placed in the container. Aigner et al. [1] proposed creating tailor-made containers by composing property-dependent non-functional services with core services (i.e., basic infrastructure services such as component instantiation and connection). Furthermore, they proposed using aspect orientation as the means to modularize the non-functional concerns. Despite these similarities, the work presented in this paper differs from theirs in that they suggest weaving the aspects together with the component code, whereas the solution in this paper keeps the container with the aspects in a binary, separate from the component core. In this aspect, the work of Duclos et al. [4] is aligned with the approach in this paper because they do not require access to the source code of the component in order to weave the aspects. However, they created a new aspect language.

JBoss [11] and the Spring framework [18] have proprietary AOP extensions that allow modularizing non-functional services in EJB. The main difference with the approach presented in this paper is that their services are implemented as interceptors which are invoked dynamically at runtime. For instance, when an advised method is called in JBoss, the list of interceptors is traversed, invoking each of them, and finally the component code is invoked using Java reflection. The technique described in this paper uses compile-time weaving and no runtime reflection, which makes it more suitable for real-time and embedded systems.

6. CONCLUSIONS

Component containers can be used to implement several non-functional concerns that would otherwise have to be implemented by the component developer. This paper made contributions in two different facets of containers. First, it showed how generative techniques can be used to generate custom stubs and skeletons, making use of AspectC++ as a compile-time reflection mechanism and as the means to compose the generated code with the component

code; and using template meta-programming for code generation. What makes this approach different from others is that the stubs and skeletons are generated without any complex infrastructure other than the AspectC++ compiler, while still being able to generate typed stubs and skeletons. The approach has limitations such as the need to use pin indexes instead of names, and that the declaration of a pin as a function still does not relieve the developer from declaring the pin in the component's introspection structures[7]. Nevertheless, the example presented demonstrated that the code of the component was greatly simplified, letting the developer focus on the core logic of the component and letting the inner container (stubs and skeletons) carry out all the repetitive but nonetheless error-prone tasks.

Second, the paper addressed the role of containers to implement non-functional concerns. In this aspect, one shortcoming of component technologies such as EJB is that they only support a predefined set of non-functional features. In order to clearly separate all the non-functional concerns from the component, containers must support the composition of non-functional features. This paper illustrated how AOP used in conjunction with containers is a viable method to achieve this goal. Although this idea is not novel [1, 4], the approach discussed in this paper showed a concrete implementation that does not require a new language or a runtime reflection mechanism. The example presented the aspect version of a container non-functional feature, namely the sporadic server. The sporadic server container had already been created for Pin using another technique [10]. However, with the technique presented in this paper, it is possible to compose other non-functional features, such as encryption, in the same container. More case-studies are needed to know if the requirement to exclusively use the interface of the container for pointcuts imposes a limitation on the kinds of non-functional concerns that can be implemented. Another open issue is about the order of composition of several features. For some feature combinations, the order is irrelevant (e.g., the sporadic server and the encryption features). However other features may require consistent and symmetric ordering as is the case of encryption composed with data compression. This is still an open research issue.

Acknowledgments

Thanks to Kurt Wallnau for the helpful discussions and feedback during the early stages of this paper. Thanks to Scott Hissam and Sagar Chaki for their reviews.

References

[1] R. Aigner, C. Pohl, M. Pohlack, and S. Zschaler. Tailor-made containers: Modeling non-functional middleware service. In *Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference*, 2004.

[2] D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel. Integration of non-functional properties in containers. In *Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[3] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[4] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, New York, NY, USA, 2002. ACM Press.

[5] A. Ferscha, M. Hechinger, R. Mayrhofer, and R. Oberhauser. A light-weight component model for peer-to-peer applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04)*, Washington, DC, USA, 2004. IEEE Computer Society.

[6] A. Gal, O. Spinczyk, and W. Schröder Preikschat. On aspect-orientation in distributed real-time dependable systems. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, Washington, DC, USA, 2002. IEEE Computer Society.

[7] M. Gonzalez Harbour and L. Sha. An application-level implementation of the sporadic server. Technical Report CMU/SEI-91-TR-026, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, September 1991.

[8] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[9] S. Hissam, J. Ivers, D. Plakosh, and K. Wallnau. Pin component technology (V1.0) and its C interface. Technical Note CMU/SEI-2005-TN-001, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, April 2005.

[10] S. Hissam, G. Moreno, and K. Wallnau. Using containers to enforce smart constraints for performance in industrial systems. Technical Note CMU/SEI-2005-TN-040, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, August 2005.

[11] JBoss Home Page. `http://www.jboss.org`.

[12] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[13] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04)*, 2004.

[14] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[15] N. Myers. Traits: A new and useful template technique. *C++ Report*, June 1995.

[16] O. Spinczyk, D. Lohmann, and M. Urban. Advances in AOP with AspectC++. In *Proceedings of the 4th International Conference on Software Methodologies, Tools, and Techniques, SoMeT'05*. IOS Press, 2005.

[17] O. Spinczyk, D. Lohmann, and M. Urban. Aspectc++: An AOP extension for C++. *Software Developers Journal*, June 2005.

[18] Spring Framework Home Page. `http://www.springframework.org`.

[19] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[20] K. Wallnau. Volume III: A technology for predictable assembly from certifiable components (PACC). Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, April 2003.

[21] K. Wallnau and J. Ivers. Snapshot of CCL: A language for predictable assembly. Technical Note CMU/SEI-2003-TN-025, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, June 2003.

[22] N. Wang, D. Schmidt, O. Othman, and K. Parameswaran. Evaluating meta-programming mechanisms for ORB middleware. *IEEE Communications Magazine*, 39, 2001.

[7] These structures contain the number of sink and source pins, their names, and their mapping to reactions.