Program Instrumentation for Debugging and Monitoring with AspectC++*

Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat
University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
{mahrenho,olaf,wosch}@ivs.cs.uni-magdeburg.de

Abstract

Monitoring is a widely-used technique to check assumptions about the real-time behavior of a system, debug the code, or enforce the system to react if certain deadlines are passed. Program instrumentation is needed to insert monitoring code fragments into the monitored system if the monitor is implemented without hardware support.

This paper describes a language-based approach to automated program instrumentation using the general purpose aspect language AspectC++. The language is an extension to the C/C++ programming language. It provides language features that allow a highly modular and thus easily configurable implementation of monitoring tasks and supports re-use of common implementations. Even though the AspectC++ language provides a convenient level of abstraction no overhead is imposed on the system in comparison to pure C/C++ code.

1. Introduction

Besides simulation and analytical modeling, measurement is one of the three evaluation techniques available for software systems. To measure the properties of a software system its internal state must be exposed. To make this internal state visible to the outside world monitoring systems are used.

There are three types of monitoring systems: software, hardware and hybrid systems. Software monitors are software extensions to an existing system. In contrast hardware monitors log the states of the different hardware buses to track the behavior of the software. Hybrid monitors take advantage of both methods - they use software extensions to generate special signal patterns on the observed buses and external hardware devices to recognize and record the events. Both approaches have their pros and cons. Software monitors on the one hand are cheap and highly flexible but they perturb the monitored system. Hardware monitors on the other hand do not necessarily interfere with the target system. But these systems highly depend on the target processor and thus are expensive and inflexible compared to a software monitor.

Especially in real-time environments it is important to use monitoring techniques. It is impossible to insure by measurement that a system is real-time capable, i.e. if it will meet its deadlines in the future. This is subject to the system design. But monitoring can assist in making statements about the time behavior of the system. If an algorithm is real-time capable it is still unknown if it works in time with a particular hardware. Monitoring should further be used to check a system design, because if you can hold certain deadlines in theory this does not guarantee the same behavior in the real-world.

In this paper we will concentrate on automated program instrumentation using AspectC++ to implement software monitors. Program instrumentation is the act of injecting monitoring code into the software system that is the subject of the monitor. Different approaches to program instrumentation are possible. It can be done by binary code transformation, link-time manipulation, special hooks in the virtual machine (in case of an interpreted language), or by source code transformation. The source code transformation approach has the following advantages:

Flexibility Monitoring is not restricted to interception of functions call. At any point in the source code monitoring code can be injected.

Abstraction-level The recorded monitoring data can have the abstraction level of programming language constructs.

Portability The instrumentation tool can be used for all target platforms that are supported by the programming language.

*This work has been partly supported by the German Research Council (DFG), grant no. SCHR 603/1-1 and SCHR 603/2.

In the following sections of this paper we will present a language based approach to automated program instrumentation by source code transformation. The main idea behind this approach is that program instrumentation by code transformation has a lot in common with compile-time aspect weaving and that monitoring is a crosscutting concern of a software system that should be implemented in a modular way by aspects. All these terms (e.g. aspect and weaving) are the vocabulary of aspect-oriented programming (AOP) [10], which will be briefly introduced in section 2.

Several languages that allow to facilitate aspect-oriented programming are available today. For instance, AspectJ[1] [9] is an extensions to the Java[2] language that combines the abstraction mechanisms of the object-oriented core language with new language features for aspect-oriented programming.

Real-world embedded and real-time systems are typically forced to get on with minimal resources in terms of memory usage and CPU speed and cannot effort the expensive Java run-time environment. Therefore AspectC++ will be the better choice in many cases. AspectC++[3] is a language extension to C/C++ that has semantically a lot in common with AspectJ but generates code that has the efficiency of C/C++. Section 3 will introduce the most important concepts of this new language.

Section 4 will then show how these language features can be used to implement an automated instrumentation process very flexibly on a high abstraction level for the most important classes of monitoring tasks. The paper ends with a discussion of related work in section 5 and our conclusions in section 6.

2. Aspect-Oriented Programming

Aspect-oriented programming tries to solve the problem that often a single dimension of functional decomposition is not sufficient to implement all aspects of a program in a modular way. This means that code that stems from a single design decision is widely spread over the system. It cannot be encapsulated in a single function, class or module. This so called *aspect code* is tangled with the normal *component code* that fits into the functional decomposition scheme. A widely used example for this effect is the synchronization code in non-sequential programs, which is also quite common in the embedded systems domain. There is a separate design decision that says where synchronization code needs to be and what the code has to do, but it cannot be encapsulated cleanly.

[1] AspectJ is a trademark of Xerox Corporation.

[2] Java is a registered trademark of Sun Microsystems, Inc. in the U.S. and other countries.

[3] AspectC++ homepage: `http://www.aspectc.org`

AOP helps out of this dilemma, because an aspect-oriented language environment allows to implement such crosscutting concerns in modular units called *aspects* and a tool – the *aspect weaver* – inserts code fragments that are derived from the aspect code to wherever they are needed. These insertion points are called *join points*.

Figure 1 shows how aspect code is woven by the aspect weaver into component code. The generated code fragments are strongly tangled, but the source code is organized in a clean modular way.

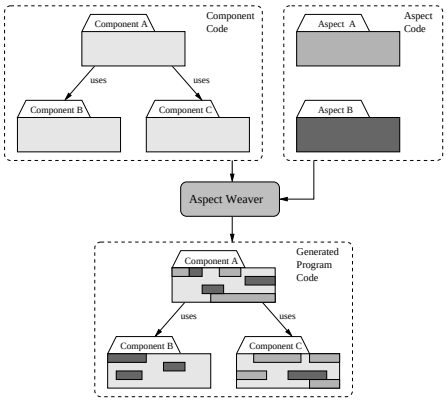

Figure 1. Weaving aspect code

Aspect weaving can be done at compile-time or at run-time. In this paper we will focus on *compile-time aspect weaving*, because the additional cost for having an aspect weaver in the target system performing run-time code injection is usually unacceptable for deeply embedded systems.

3. AspectC++

AspectC++ [13] is an extension to the C/C++ programming language. It's aim is to support aspect-oriented programming even in domains where resource limitations do not allow to use modern but more expensive languages. This section will introduce the basic concepts of this new language.

3.1. Pointcuts

In AspectC++ join points are defined as points in the component code where aspects can interfere. A join point refers to a method, an attribute, a type (*class*, *struct* or *union*), an object, or a point from which a join point is accessed.

A *pointcut* is a set of join points described by a *pointcut expression*. Pointcut expressions are composed from *match expressions* used to find a set of join points, *pointcut functions* used to filter or map specific join points from a pointcut, and algebraic operators used to combine pointcuts.

Match expressions are strings containing a search pattern. AspectC++ can match type names as well as attribute and method signatures. The wildcard symbol "%" allows flexible searches. Figure 2 shows examples for match expressions.

types:	matches . . .
"int"	the built-in scalar type *int*
"Memory%"	all classes, structs or unions having a name starting with *Memory*
attributes:	
"Chain* Chain::next"	the attribute *next* of the class *Chain*
"% State::%"	all attributes of any type of the class *State*
methods:	
"int main(int, char**)"	the function *main* having exactly the given signature
"% printf(. . .)"	*printf* with any number and types of arguments and any result type
"void %(int, %)"	functions with any name, taking an int as first argument, and having a second argument of any type
"void %::clear()"	the *clear* methods without arguments of any class
"void Mode::set%(. . .)"	all methods of *Mode* having a name that starts with *set*

Figure 2. Examples for match expressions

Match expressions alone are not sufficient to clearly identify a join point. For example, consider a method match expression. It will match not only all implementations of matching methods, but also all calls to those methods. To filter specific types of join points AspectC++ supports pointcut functions. In figure 3 the most important built-in pointcut functions of AspectC++ are listed.

To support reuse of complex pointcut expressions AspectC++ allows to define named pointcuts. A named pointcut can contain formal arguments, which represent context exposed from the join points. This context information can be used by the aspect code, which is inserted at or called from the join point location. The following example shows the definition of a named pointcut `IRQ_level_call`. It refers to all calls of the function `void IRQ::level(int)` and exposes the single integer argument of that call.

```
pointcut IRQ_level_call(int irq_level) =
  call("void IRQ::level(int)") && args(irq_level);
```

Pointcuts are the key language element to deal with the crosscutting nature of aspects. They can describe points in the static structure or the dynamic control flow of a program in a very flexible way. There is no restriction to a specific function, class, module, or other unit of functional decomposition.

methods:	
call(*pointcut*) execution(*pointcut*) callsto(*pointcut*)	*call* filters all join points that are method calls while *execution* extracts those join points referring to the method implementation. *callsto* gives all corresponding call join points to the execution join points in the argument pointcut.
attributes:	
get(*pointcut*) set(*pointcut*)	*get* and *set* select any attribute access join points from the *pointcut*.
types:	
classes(*pointcut*) derived(*pointcut*) base(*pointcut*) instanceof(*pointcut*)	*classes* delivers only the classes, structures, and unions from a join point. With *derived* it is possible to refer to all types in the pointcut expression and all classes derived from them. *base* can be used to find all base types of classes in a pointcut while *instanceof* can be used to locate objects of certain types.
scope:	
within(*pointcut*)	*within* derives all join points declared in methods of types in the *pointcut*.
control flow:	
cflow(*pointcut*) reachable(*pointcut*)	*cflow* captures join points occurring in the dynamic execution context of join points in the *pointcut*. *reachable* results in all join point from which the argument join points can be reached.
context:	
that(*type pattern*) target(*type pattern*) args(*type pattern*, . . .)	*that* returns all join points where the current C++ *this* pointer refers to an object which is an instance of the type described by *type pattern*. *target* finds all join points where the target object is an instance of the type in *type pattern*. *args* filters all methods or attributes with a matching signature.

Figure 3. Pointcut functions in AspectC++

3.2. Advice

An advice declaration can be used to specify code that should run when the join points specified by a pointcut expression are reached:

```
advice IRQ_level_call(irq_level) : void after(int irq_level) {
  cout << "Interrupt level set to " << irq_level << endl;
}
```

With this advice each call to `IRQ::level(int)` is followed by the execution of the advice code, which prints the number of the new interrupt level.

Different kinds of advices can be declared, including *after* advice that runs after the join point, *before* advice that is executed before the join point, and *around* advice, which is

executed in place of the join point. *Around* advice can explicitly request the execution of the original join point code using `proceed()`:

```
advice IRQ_level_call(irq_level) : void around(int irq_level) {
  cout << "The interrupt level is about to change." << endl;
  proceed();
  cout << "Interrupt level set to " << irq_level << endl;
}
```

If the advice is not recognized as being of a predefined kind, it is regarded as an introduction of a new method or attribute to all join points contained in the pointcut expression. In this case the pointcut expression must only contain join points of the type class.

```
pointcut all_classes() = classes("%");
advice all_classes() : void print() {
  cout << "Address: " << (void*)this << endl;
}
```

To obtain more information about the current join point, the advice code can use the object referenced by `thisJoinPoint`. The methods defined on `thisJoinPoint` provide access to static join point attributes like a string representation of the join point, the argument types, a unique ID, and the location of the join point in the source code as well as a dynamic part, which contains for example the current argument values of a call join point. This language feature allows to implement generic advice code. For instance, advice code can iterate over the arguments of a function that is unknown at implementation time. Possible applications are execution trace aspects or argument marshalling for a remote method invocation.

Even introductions can use `thisJoinPoint`. For example the following code fragment extends the introduction already presented above with printing of the class name.

```
advice all_classes() : void print() {
  cout << "Address of " << thisJoinPoint->toString ()
       << " object: " << (void*)this << endl;
}
```

AspectC++ generates members of the object referenced by `thisJoinPoint` overhead-free on demand by analyzing the advice code.

3.3. Aspects

While named pointcut declarations can appear everywhere where declarations are allowed, advices can only be defined inside an *aspect declaration*. Aspects in AspectC++ implement in a modular way crosscutting concerns and are an extension to the class concept of C++. Additionally to attributes and methods, aspects may also contain advice declarations. This allows advice code to preserve state information. Several examples of aspect implementations will follow in the next section.

AspectC++ offers virtual pointcuts and aspect inheritance to support the reuse of aspects. A virtual pointcut can be redefined in a derived aspect and the inherited advice will use the new pointcut definition. Pointcuts can also be pure virtual. In this case the pointcut definition has to be overwritten by a derived aspect before it can be instantiated. Aspects can also inherit from a class, but it is not possible to derive a class from an aspect.

4. Program Instrumentation

The introduction section already mentioned that there are different forms of program instrumentation. This section concentrates on the the main uses of program instrumentation: debugging, profiling/measurement, and runtime surveillance/monitoring.

When a program crashes or delivers wrong results, programmers often add print statements to the functions involved to trace the program flow or the steps of a computation. This can also be done using an automated program instrumentation with AspectC++. The major benefit of using aspects is that the program source need only be touched for corrections or improvements after the bug has been found. This avoids the risk of screwing things up by accident, e.g. when removing the print statements.

The second use for program instrumentation is profiling and measurement. Both are used in the evaluation phase of the software development cycle. Profiling is referred to as the process of collecting data about the way a program or parts of it including the appropriate resources are used during runtime. Examples are the rate of occurrence of interrupts of the network driver or the number of calls to different functions in the system. Knowing these numbers gives an advice on parts of the system that are worth to be optimized to improve the overall system performance. It makes less sense to optimize a function that nearly never gets called when there are several other functions that account for almost all the processing time. Measurement in contrast gives absolute numbers on the time spend in a function or the resources used by it. These numbers can be used to compare two implementations or one implementation with a specification or requirement to find the most suitable version or to decide that it needs to be reworked because it is to slow to guarantee a dead line in a real-time application.

The last use mentioned above is runtime surveillance and monitoring. They are closely related but we distinguish them in the way that monitoring collects data at runtime to make them visible to the outside where runtime surveillance collects data to compare them with predefined values to react in certain ways when they reach a threshold.

In the next subsections we will show three instrumentation examples for measurement, monitoring/profiling and

runtime surveillance using AspectC++.

4.1. Example 1: Monitor Task Switches

The first examples shows how to monitor the context switches of the PURE[4] operating system. The system is highly configurable, so the aspect can not exactly know, which classes, interrupt handlers and so on will call the scheduler. Also the internal structure of the scheduler varies. The only thing we know about all possible configurations is, that the innermost scheduler function is `Coroutine::resume`, so the aspect code can only use this knowledge.

Figure 4 shows the aspect code used to perform this instrumentation. Line 1 includes the header file for the sensors of the monitoring system. The lines 2-13 define the actual aspect. The first step is the definition of two pointcuts – the first (`switch`) contains the execution join point of the known method, the second (`caller`) contains all execution join points from where the first pointcut can be reached by direct or indirect calls. Indirect calls are calls to functions that itself call the target function directly or indirectly. The lines 7-12 define an around advice that replaces all join points in the pointcut `caller` with the given code fragment. This replacement code instruments all functions leading to the target function. It first calls a sensor that logs the entry to the function, than executes the original function body using `proceed()`, and last calls a sensor to log the completion of the function. We use the `thisJoinPoint` object and its unique number to identify the sensor and the events it generates in the event trace.

```
1  #include <Sensors.h>
2  aspect MonitorContextSwitch
3  {
4    pointcut switch() = execution("void Coroutine::resume()");
5    pointcut caller() = execution(reachable(switch()));
6  public:
7    advice caller() :
8      void around() {
9        Sensor::Log_entry(thisJoinPoint->id());
10       proceed();
11       Sensor::Log_exit(thisJoinPoint->id());
12     }
13 };
```

Figure 4. Monitor context switches

On a Pentium system this instrumentation introduces (in addition to the sensor code [11]) an overhead of 40 byte code and 4 byte data per join point. Most of this code is only used for the static initialization of the context information (unique join point id) stored in the `thisJoinPoint` object. So the additional code including the sensor code resulted in an average runtime overhead of 200 ns to 1000 ns per execution (depending on the sensor configuration). Cache effects and the memory speed have a significant influence on this – but it always has an upper bound.

The figure 5 shows the measured results for a two-thread configuration where each of the threads just calls the scheduler to switch to the next. The detailed view shows a single switch from thread 1 to 2. The vertical dashed and dotted lines mark the start and end of `Coroutine::resume`. The numbers[5] on the timescale are processor cycles relative to the first sensor event. With such a view developers are able to identify inefficient implementations on this (time) critical execution path.

```
...
pointcut caller() = within(classes("Coroutine" ||
                                   "Actor" ||
                                   "Activity")) &&
                    execution(reachable(switch()));
...
```

Figure 6. Restricted pointcut definition

One possible problem with this special instrumentation is the possibly large number of inserted sensors which may extend the monitored system above the available memory – which is often tight especially in deeply embedded systems. One possibility to reduce the number of sensors is to restrict the instrumentation to only some classes by extending the second pointcut definition with the `within` declaration. The modified source is shown in figure 6. A second possibility is to optimize and streamline the sensors inserted into the system – which is described in [11].

4.2. Example 2: Monitor the Memory Management System

In the second example we show how to monitor the creation and destruction of objects in the system. There are two reasons for doing this. First we try to find memory leaks. Because nearly nobody uses a garbage collector in C/C++ developers have to care about the allocated memory themself. Second we like to know more about how the application uses the memory. Maybe it often allocates and destroys objects of the same class or it allocates and destroys a large number of objects within a loop. If we know of such a behavior it would be possible to optimize the memory management for this special cases or to transform the application to use a special memory manager within certain regions of the code.

Figure 7 shows the source code for this aspect. In this aspect we use a virtual pointcut definition. So the aspect does not gets instantiated unless the virtual pointcut is overloaded with a concrete one in a derived aspect. The lines 20-22 and 23-25 give two examples for derived aspects. The first selects only the class `Foo`, the second all classes for the pointcut. This technique can be used to build some kind of a

[4] PURE [1] is an object-oriented operating system family that mainly targets the area of deeply embedded systems. It is a development of our group.

[5] This times are measured on a 75MHz Pentium system.

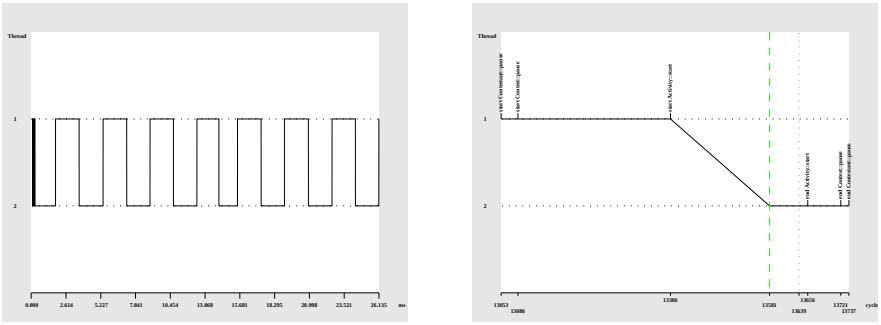

Figure 5. Context switches (overview, detailed)

```
1   #include <Sensor.h>

2   aspect Instances {
3     pointcut virtual pClasses () = 0;
4     advice pClasses () :
5       class Counter {
6         static int counter;
7       public:
8         Counter () {
9           counter++;
10          Sensor::Log(thisJoinPoint->id(), counter);
11        }
12        ~Counter () {
13          counter--;
14          Sensor::Log(thisJoinPoint->id(), counter);
15        }
16      } __counter;
17  };

18  advice Instances::pClasses () :
19    int Instances::Counter::counter = 0;

20  aspect MonitorFooInstances : public Instances {
21     pointcut pClasses () = classes ("Foo");
22  };

23  aspect MonitorAllInstances : public Instances {
24     pointcut pClasses () = classes ("%");
25  };
```

Figure 7. Monitor object lifetimes

”Common Debugging Aspects Library” that includes a collection of predefined debugging aspects that only have to be applied to a set of classes when needed.

This aspect works very simple. It adds an inner class and an attribute of this class to all selected classes in the system. This inner class contains a static attribute to get a per-class counter variable. Every time an object of the instrumented class gets instantiated the constructor of this inner class gets called and increases the instances counter. The constructor also calls a sensor to record the time and the instances counter. When the object is destroyed also the destructor for the inner class is called. It then decreases the instances counter and calls a sensor to record this event. This data can later be used to analyze and evaluate the memory usage of the application. To initialize the static class attribute of the inner class we use a second introduction (line 18,19). This adds a global, initialized variable for each instrumented class.

Figure 8 shows the results of a sample measurement[6]. For this profiling task we instrumented a class that buffers

[6]This measurement has been done on a 66MHz PowerPC 823.

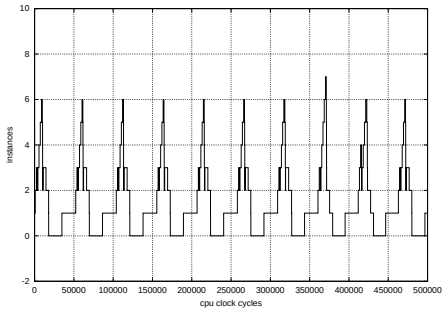

Figure 8. Instances example results

incoming data between a device driver and the application layer that works asynchronous to the driver. This diagram shows several things: first that we do not forget to free any objects and second that the messages arrive in a regular pattern and are consumed by the application in a burst. It also shows that these objects are created and destroyed on a high rate (over 1000 per second) so a specialized memory manager can noticeable improve the overall system performance.

4.3. Example 3: Add Run-time Surveillance

The third example shows how to use AspectsC++ for runtime surveillance. In a real-time environment a correctly working component not only means to perform the specified functions but also to complete them in a predefined time. So using general purpose components that are not developed with a real-time use in mind, are not guaranteed to work properly in such an environment. A way to make them usable is to add a timeout to force a bounded execution time. The figures 10 and 11 depict the two possible scenarios. Either the monitored call completes within the given time bounds or the watchdog timer aborts it and returns to the caller or activates an exception handler. Whether the incomplete execution is acceptable (e.g. printing a debug message) or not (e.g. storing data on a hard disc) has to be decided by the system designer.

```
1  aspect GuardedExecution {
2    pointcut guarded() =
3      executions("void Log::output(const char*)") &&
4      cflow(executions("void doCriticalRequest()"));
5   advice guarded() : void around() {
6      guard.enter(10); // 10ms
7      proceed();
8      guard.leave();
9    }
10 };
```

Figure 9. Guarantee bounded execution times

Figure 9 shows the aspect code used for this example. We first define a pointcut and select the execution of `Log::Output(...)` for it. We additionally use the `cflow` pointcut designator to select only those join points which appear within the dynamic execution context of the method `doCriticalRequest()`. So the aspect only applies if the request was issued from a control flow originated in `doCriticalRequest()`. So the output method can still be called with their original timing behavior, which means a possibly infinite execution time. This technique is not only useful in a real-time environment but can also be used to add a timeout feature to functions in a black box component.

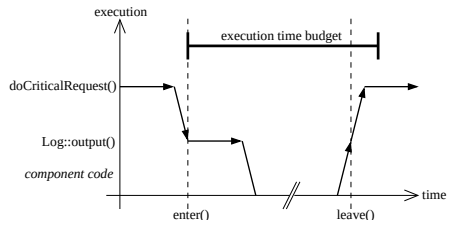

Figure 10. Call completed within time bounds

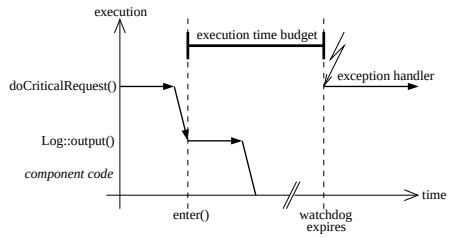

Figure 11. Call exceeds time bounds

5. Related Work

Using aspect-oriented implementation techniques in conjunction with C++ does not seem to be very popular in the AOP community. Only very few contributions related to C++ can be found in the proceedings of relevant conferences and workshops of the last years. We explain this with the overall "Java hype" and, more than that, with the lack of tool support. For instance, L. Dominick describes "lifecycle control aspects when applying the Command Processor pattern" and complains "because no weaver technology was available, C preprocessor macros were used" [4].

A very interesting approach is followed by FOG [14]. FOG is a meta-compiler for C++ supporting a superset of the language. Similar to the AspectC++ implementation it is a source to source translator, but the language concept differs. In FOG the C++ "One Definition Rule" is replaced by a "Composite Definition Rule". This allows, for instance, to define multiple classes with the same name, which FOG will then merge into a single class. Functions and attributes can be easily added this way to classes. Function code can be extended with a similar mechanism. While FOG seems to be ideally suited for subject-oriented programming [6][12] the join point model is much less powerful in comparison to AspectJ/C++. Especially the algebraic operations on "pointcuts" and the notion of control flow are useful in many aspect implementations.

More powerful than the FOG approach is OpenC++ [2]. It allows a compiled C++ meta-program to manipulate the base-level C++ code. Classes, types, and the syntax tree of the base-level are visible on the meta-level and arbitrary transformations are supported. OpenC++ allows aspect-oriented programming, but language extensions that especially support AOP are not provided.

AspectC [3] is an aspect-oriented extension to plain C, which is currently under development to study crosscutting concerns in operating system code. It is not planned to support C++ component code with this implementation [8]. AspectC also adopts the key concepts from AspectJ, but the non-object-oriented nature of C forces AspectC to leave out many useful features like using inheritance to compose aspects. As C is basically a subset of C++ the AspectC++ language extension can be used with C as well.

The need for program instrumentation especially for measurement and monitoring exists for a long time and there are some solutions that should be mentioned. The first is the MC4P tool of the JUWEL system [5] that can be used for the instrumentation of C++ code to monitor method calls and the status of object attributes after such a call. The second solution is the MDL language and its compiler, which is part of the Paradyn Parallel Performance Tools [7]. The *Metric Description Language* (MDL) is used to specify points in the program (call statements; procedure entry, exit) where special code has to be inserted to calculate the metric and a binary rewriter for the instrumentation. Both systems use a concept similar to the join point model of AspectC++, but the possible join points are limited in comparison. In addition both systems are solely designed for measurement and monitoring in a specific context and not intended to implement highly reusable debugging and

monitoring code.

6. Conclusions and Future Work

In this paper we have introduced AspectC++ as a general purpose language for aspect-oriented programming with C/C++ and have demonstrated its usefulness and efficiency for debugging and monitoring tasks by giving typical examples. Although it is in an early development phase, the language can already be used for a wide range of applications (e.g. all the presented ones). AspectC++ enables to develop reusable aspects by providing features like aspect inheritance and virtual pointcut definitions. This allows a clean separation of debugging and monitoring code from the component code. The aspect code takes over a mediator role between both as it is illustrated in figure 12.

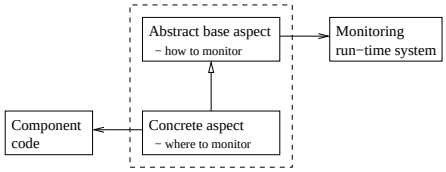

Figure 12. The mediator role of aspects

The semantics of the AspectC++ language is closely related to AspectJ. Thus developers that are familiar with AspectJ and C/C++ can benefit from their experiences. AspectC++ is especially interesting for the deeply embedded systems area, because in this domain C/C++ is still the dominating programming language and the number of tools that allow aspect-oriented programming with C/C++ is minimal. AspectC++ and a set of working code examples is available for public download from `http://www.aspectc.org/`.

Future work will involve the implementation of missing (planned but not mentioned in this paper or yet unknown) AspectC++ language features as well as improvements to the underlying PUMA framework[7] (our C++ code transformation system). We will extend it to handle more non-standard compiler specific C/C++ extensions to make it usable for large real-world projects and experiment with complex application scenarios.

References

[1] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE Internation Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St. Malo, France, 1999. IEEE Computer Society.

[2] S. Chiba. Metaobject Protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, Oct. 1995.

[3] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring Operating System Aspects. In *Communications of the ACM*, pages 79 – 82, Oct. 2001.

[4] L. Dominick. Aspect of Life-Cycle Control in a C++ Framework. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, Lisbon, Portugal, June 1999.

[5] M. Gergeleit. *Automatic Instrumentation of Object-Oriented Programs*. Technical report, German National Research Center for Information Technology, 1994.

[6] W. Harison and H. Ossher. Subject-Oriented Programming (a Critique on Pure Objects). In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 411–428, Woshington, D.C., Sept. 1993. ACM.

[7] J. K. Hollingsworth et al. *MDL: A Language and Compiler for Dynamic Program Instrumentation*. Technical report, Computer Sciences Department, University of Maryland; Computer Sciences Department, University of Wisconsin, May 1997.

[8] G. Kiczales, July 2001. Personal communications.

[9] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCS*. Springer-Verlag, June 2001.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[11] D. Mahrenholz. Minimal Invasive Monitoring. In *Proceedings of: The Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, May 2001.

[12] H. Ossher and P. Tarr. Operation-Level Composition: A Case in (Join) Point. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium, July 1998.

[13] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of *Conferences in Research and Practice in Information Technology*. ACS, 2002.

[14] E. D. Willink and V. B. Muchnick. Weaving a Way Past the C++ One Definition Rule. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, Lisbon, Portugal, June 1999.

[7] PUMA homepage:
`http://ivs.cs.uni-magdeburg.de/~puma/`