An application of dynamic AOP to medical image generation

Thomas Fritz[a,1], Marc Ségura[b], Mario Südholt[b], Egon Wuchner[c], Jean-Marc Menaud[b]

[a]Institut für Informatik, Gruppe PST
Ludwig-Maximilians-Universität
Oettingenstr. 67
80538 München, Deutschland
`fritz@informatik.uni-muenchen.de`

[b]Équipe OBASCO
EMN-INRIA, LINA
École des Mines de Nantes
4, rue Alfred Kastler
44307 Nantes cedex 3, France
`{msegura,sudholt,jmenaud}@emn.fr`

[c]Corporate Technology, SE2
Siemens AG
Otto-Hahn-Ring 6
81739 München, Deutschland
`Egon.Wuchner@siemens.com`

ABSTRACT

Medical image generation, *e.g.*, in computer tomographs, requires the use of sophisticated algorithms in a highly sensitive application domain. These algorithms are characterized (i) by a large variability to enable generation of different types of images and (ii) a strong need for dynamic reconfiguration to adapt image generation to individual patients. These two characteristics suggest the use of AOP techniques to manage variability which is akin to a crosscutting functionality and to enable dynamic reconfiguration.

In this paper we present three results related to AOP and medical imaging in the context of medical devices from Siemens AG, Germany: (i) a motivation why imaging software for medical tomographs can benefit from dynamic AOP, (ii) a case study of how system software for medical devices can be adapted using the Arachne system for dynamic AOP in C, and (iii) a detailed presentation of the underlying Arachne implementation and the design of its extension to C++.

1. MOTIVATION: MEDICAL IMAGE GENERATION AND AOP

Many medical devices (*e.g.*, magnetic resonance or computer tomography devices) require the generation of images based on measurements from the human body. The corresponding signal processing consists in the decomposition of the input signals yielded at certain points of time into signals corresponding to all the positions within a space cube representing the scanned 3-dimensional image (see Fig. 1). The signal specific to a particular position within the cube is characterized by its periodicity (*i.e.* cosine and sine waves),

---

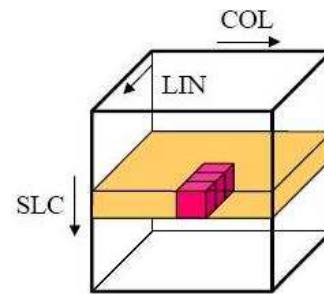[1]Part of this work has been done during the author's stay at École des Mines de Nantes.

Figure 1: Space cube of measured signals associated with a human body part

frequency, phase and amplitude.

The mathematical tool underlying such image generation tasks is Fourier transformation. In medical scanners, measurement data is typically stored within a raw data cube consisting of lines, columns and slices. A position in this cube is determined by its three dimensions and represents one measured signal. On the other hand this cube also represents the part of the human body which is examined. The state of human tissue at a position, is calculated by application of a sequence of basic image calculation steps to the measured raw data. These steps filter and adjust received signals, calculate images and post-process images. In the existing software for devices of Siemens AG the entire image generation transformations are constructed from approx. 60 different basic image processing functors. The set of valid transformations, *i.e.*, valid orderings according to which these basic functors may be combined, can be conceptually represented using a graph with one start and end node. The start node receives the measured signals and the end node yields a generated image.

In practice, the devices are used as follows. A concrete transformation needs to be configured before the start of a measurement for a patient. Currently, doctors execute one of a set of complete functor sequences generating an image for a patient, followed by other complete sequences, if necessary, for the same patient. However, based on corresponding customer requests, an evaluation is performed within Siemens medical devices unit of explicit support for

dynamic adaptations of such functor sequences. With such techniques medical staff would be able to interactively adapt image generation during a measurement session depending on an initial set of calculated images. This would be highly useful in order to optimize the final images *w.r.t.* the individual patient. Such adaptations would enable, *e.g.*, using a higher resolution for parts belonging to tumors and are expected to speed up generation of the images taken for each patient.

During execution, code is executed corresponding to sequential and parallel functor sequences, the latter implementing, *e.g.*, calculations of different parts using different resolutions. This means execution can be represented by another graph, henceforth called the functor graph, which is a subgraph of the graph of all valid transformations discussed above.

Adaptation of functor sequences constitutes a software engineering problem that has three main characteristics:

1. The changes required by these adaptations are *scattered* over the functor graph and require a partial, but possibly rather comprehensive, transformation of the original processing graph.

2. The modifications to the functor graph during a measurement *cannot be anticipated*.

3. Modifications to the functor graph must be *reversible* so that new measurements can be performed based on parts of the image information previously generated.

Aspect-Oriented Programming (AOP) [11] has been instrumental in the adaptation of complex legacy software (for an example in the domain of operating systems programming, see [1]). An application of AOP techniques for the implementation of such medical image generation software seems therefore promising. In particular, an AO approach realizing this adaptation problem through (relatively) small changes to an existing code base seems advantageous, *e.g.*, concerning development effort and correctness validation, compared to approaches incurring larger changes, such as restructuring of the code base into an interpreter over the functor graph.

In this paper we present initial results of how to address the adaptation problems for image generation software for medical scanning devices from Siemens AG. We show how to apply the Arachne model and tool [6] for dynamic AOP in C in order to directly address the three above-mentioned characteristics: Arachne enables (i) the concise modular definition of the changes to the functor graph, (ii) dynamic modification of functor graphs without access to the source code, and (iii) unweaving of functor modifications. Furthermore, we give an overview of Arachne's implementation as well as its on-going extension to C++.

The remainder of this paper is structured as follows. Section 2 presents examples of the C++-based legacy code base used for a medical device from Siemens AG. In Sec. 3, we detail two fundamental transformations of the functor graph used for adaptation of image generation. Section 4 shows how such manipulations can be defined using Arachne. In Sec. 5 we give an overview of the architecture of Arachne and present its on-going implementation in C++. In Sec. 6 related work is discussed. Finally, Sec. 7 gives a conclusion and presents future work.
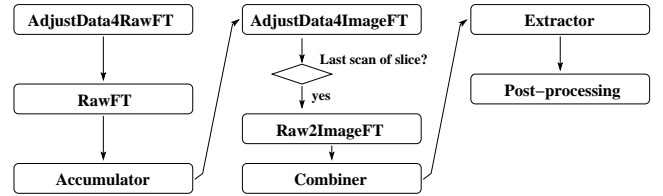

Figure 2: Basic steps for image generation

2. IMAGING CODE BASE

Let us first have a closer look at parts of the C++ code base used for medical image generation in Siemens devices, its overall structure and some specific generation steps.

Fig. 2 shows a sequence of basic processing steps for image generation. The functor `AdjustData4RawFT` (as well as the functors `Accumulator` and `AdjustData4ImageFT`) receives a line of measured data and makes some adjustments for the following Fourier transformations. The Fourier transformation `RawFT` works on the columns of the current line measurement of the current slice and derives some intermediary values for each column per receiver channel. (A channel corresponds to, *e.g.*, sensors situated at different locations of the medical device and receives its own signal during measurement.) The functor `Raw2ImageFT` takes the values of all these line calculations and computes a signal consisting of frequency and amplitude for each matrix position of the current slice. This way an image per receiver channel is calculated. The `Combiner` functor then takes the computed slice images corresponding to several receiver channels and calculates a weighted combination of them. The functor `Extractor` converts the complex values making up the image into corresponding human-readable information (*e.g.*, amplitude information) allowing conclusions about the kind of human tissue. Finally, `Post-processing` performs graphical manipulations, such as coloring of image parts, to the generated image.

The implementation of the image generation algorithms forming the basic steps are based on the cube containing raw data measurements introduced previously. On the code level, this cube does not have only three spatial dimensions but, in fact, up to 16 dimensions. For instance, its fourth dimension consists of the above-mentioned channels. Slice images can be calculated for each channel and combined afterwards. The following statement constructs such a multidimensional cube:

```
RawCube* cube =
  CubeFactory::create( LINE, 256, COLUMN, 256,
                       SLICE, 512, CHANNEL, 8,
                       ...);
```

Similarly, there is an `ImageCube` class allowing to store a multi-dimensional array of (intermediate or final) images.

Functors essentially implement algorithms iterating over the multi-dimensional data cubes. Each generation step requires access to data from a certain set of dimensions. As an example, the functor `RawFt` accesses the current line and slice and executes the Fourier transformation on each column for each receiver channel by initializing an iterator accordingly (see the iterator `rawFtIter` in lines 20–23 of List. 1).

2

```
1   class RawFT : public Functor {
2   public:
3     void addNextFunctor( Functor* );
4     FunctorList* getNextFunctors();
5     ...
6
7     virtual void computeScan( CtrlInfo& ctrl, CubeIterator& iter );
8   };
9
10  void RawFT::computeScan( CtrlInfo& ctrl, CubeIterator& iter ) {
11    // copy input iterator referring to raw data cube
12    CubeIterator rawFtIter( iter );
13
14    // set the cube dimensions and ranges the RawFT should work on
15    rawFtIter.init( COLUMN, 0, ctrl.getNumberOfColumns(),
16                    LINE, ctrl.getCurrentLine(), ctrl.getCurrentLine(),
17                    SLICE, ctrl.getCurrentSlice(), ctrl.getCurrentSlice(),
18                    CHANNEL, 0, ctrl.getNumberOfChannels());
19
20    // call RawFT
21    Imager::FT( iter, rawFtIter );
22
23    // call next Functors
24    for(int i, i<FunctorList.size(), i++){
25      this->getNextFunctor(i)->computeScan( ctrl, iter );
26    }
27  };
```

Listing 1: RawFT implementation skeleton

Functors inherit from a base class `Functor` as shown for `RawFT` in List. 1. This functor provides public methods `addNextFunctor` and `getNextFunctors` (lines 3, 4) to manage a list of functors following `RawFT` within a sequence of generation steps forming a transformation. These methods realize the functor graphs at runtime. The algorithm represented by the functor is implemented by the method `computeScan` (lines 7, 10–27). This methods first initializes the iterator `rawFtIter` and applies it (lines 12–21). Finally, all functors following `RawFT` in the current functor graph are called using the method `getNetFunctors` (lines 24–26).

3. ADAPTATION SCENARIOS

We now present two fundamental adaptations of the imaging process required to enable interactive control by the medical staff. Technically, these adaptations take the form of transformations of the graph defining the functor sequences which generate images from raw data.

The first scenario for interactive adaptation of the image generation process consists in *adjoining a new parallel functor chain* to an existing chain of the current functor graph, as illustrated by Fig. 3 (which shows an application of a transformation to a graph consisting of two parallel functor sequences). For instance, a doctor using a tomograph could find some indication of a tumor covering some part of the human body which is currently being scanned. Thus, he decides to examine the corresponding region further without loosing the currently calculated image information and without interference with other image parts. This can be done by adding a new sequence of functors performing a very detailed image calculation for the smaller body section in addition to the original calculation of the initial body part. Both chains are then executed in a pseudo-parallel fashion and the resulting images of both functor chains are
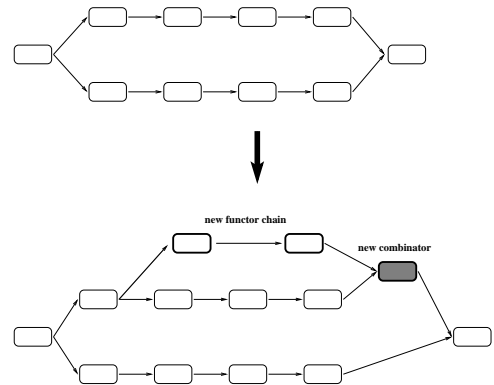

Figure 3: Adding a functor chain in parallel

combined to one image per slice.

The second adaptation scenario is to *replace a part of a functor chain* by another one as shown in Fig. 4. This scenario is used, *e.g.*, to modify the section of an image a transformation is applied to.

To conclude the discussion of image processing adaptations, note that functors affect several connection points of the original chain as illustrated by the two transformations above. Furthermore, many adaptations are performed during a tomography examination. Such adaptations may be applied to the initial functor chain as well as to chains which have been dynamically added previously. As a consequence the corresponding transformations are spatially and temporally scattered over the entire functor graphs.
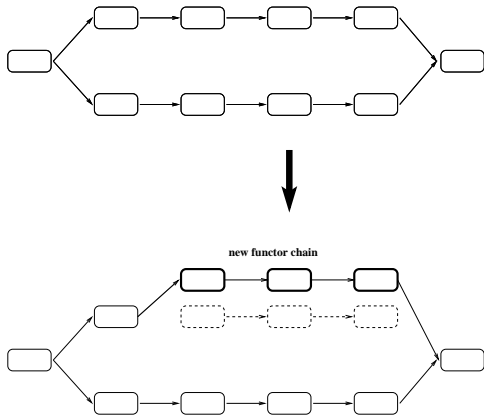
4. APPLYING DYNAMIC AOP

Figure 4: Replacing a functor chain

We now turn to the problem how to express the adaptation scenarios using the aspect language of Arachne.

Let us consider a sequence of functors, where each functor implements a method `computeScan`, as introduced in the previous section. Since the current version of Arachne supports only C, the functors have been mapped from C++ to C for the purposes of the evaluation presented in this paper. Note that this is quite simple because the functors, as exemplified by the code shown in the previous section, do not make extensive use of the object-oriented features of C++. As shown in List. 2 functors are mapped to C code by representing each functor class as a structure containing a pointer to a list of subsequent functors in the functor graph and a pointer to a function `computeScan`.

Arachne's aspect language. Let us first briefly consider Arachne's aspect language (see [6] for a more detailed presentation). Arachne provides analogues for C to AspectJ's main Java-oriented [10] features: pointcuts can be used in Arachne to match calls to C functions and match nested calls on the execution stack, a form of "cflow". (Note that Arachne also provides pointcuts others than those related to calls, *e.g.*, variable access join points. Since these are not used in this paper, we refer the interested reader to [6].)

Arachne distinguishes itself from most aspect languages by providing a construct for explicit sequencing on the language level, which is of the following form:

seq(*Prim*, *Prim* [*], ..., *Prim* [*], *Prim*)

where *Prim* is a primitive aspect, such as

call(m(a, b)) then m'(b);

associating a primitive pointcut, which matches, *e.g.*, a call, to an action, typically a function call, which is to be executed instead of the matched call and that can itself call the original function. The construct is executed by first creating a new instance of the sequence aspect (with a fresh state) each time the first primitive aspect in the sequence matches. Then the other primitive aspects of the sequence are applied (repeatedly in case a star is used) as early as possible, *i.e.*, a primitive aspect a has priority over its predecessor: a is executed when the pointcut of a matches.

Adjoining a new functor sequence. A new functor sequence can be adjoined to an existing one using Arachne's aspect language by means of a single sequence aspect. Assume we want to add a chain of functors (`f3'` to `f4'`) along with a new functor combining the data of the parallel chains to a chain of functors `f1` to `f6`. An aspect that adds the new chain at functors `f3` and `f6`, and that can also be unwoven without any side effects is presented in List. 3.

The `computeScan` functions of `f1` and `f2` are executed as usual, but once `f3` is reached (lines 6–8 in List 3) in this sequence, the new subchain will be executed and the resulting data is stored temporarily. Then the original chain is executed and before `computeScan` of `f6` is executed, the image data is combined (lines 9, 10). The second and third step in the sequence aspect contain if-conditions to ensure that the `computeScan` methods work on the same image and iterator. This is necessary because there might be several identical chains to be matched that work on different iterators.

Replacing a functor sequence. Replacement of a functor sequence by another one can be expressed using a single sequence aspect, too. Assume we want to replace functors `f3` to `f5` with new functors `f3'` to `f5'` (cf. Fig. 4). The aspect in List. 4 achieves the replacement.

The `computeScan` functions of `f1` and `f2` will be executed as usual, but once `f3` is reached, the new subchain will be executed and the `computeScan` function of `f5'` will call the one of `f6` that then proceeds as usual. As in the preceding aspect, we use if-conditions to ensure that the steps in the sequence work on the same image and iterator.

Using Arachne, the functors replacing the ones in the original chain can be added dynamically.

5. ARACHNE: ARCHITECTURE AND IMPLEMENTATION

In this section we describe how Arachne enables dynamic weaving and unweaving of aspects into running legacy C applications. We present, in particular, a description of Arachne's structure which is improved w.r.t. the one in [6], a new detailed discussion of the consistency of Arachne's rewriting of native code, and a design of Arachne for C++ developed in order to apply Arachne (as future work) to Siemens AG's original code base, which has been developed over 8 years and whose use without modification is an important cost criterion for Siemens. By targeting Arachne directly to C++ code we obviate the need for the mapping to C introduced in the preceding sections.

5.1 Arachne's Architecture

Arachne's architecture is shown in Fig. 5: it is composed of three parts: the aspect runtime environment, a kernel manager, and an aspect compiler.

5.1.1 Arachne's runtime environment

The main component of Arachne's runtime environment is *Arachne's kernel dynamic link library* (DLL). As it is responsible for weaving aspects in the base program at runtime, it has to be able to rewrite the binary code of the base program and thus needs to be loaded in the same address space (as discussed in Section 5.1.2). Once the kernel DLL is loaded in the address space, it creates a thread in the base program

```
1  typedef struct functor *Functor;
2  struct functor{
3      Functor* nextFunctors;
4      void (*computeScan)(CtrlInfo* ctrl, CubeIterator* iter);
5  }
6
7  void computeScan_RawFT(CtrlInfo* ctrl, CubeIterator* iter){...}
8  Functor rawFT; rawFT.computeScan = computeScan_RawFT;
```

Listing 2: Mapping RawFT to C

```
1  CtrlInfo* ctrlNew; CubeIterator* iterNew;
2
3  seq(call(void computeScan_f1(CtrlInfo*, CubeIterator*)) && args(ctrl,iter);
4      call(void computeScan_f2(CtrlInfo*,CubeIterator*)) && args(ctrl2,iter2)
5       && if(ctrl == ctrl2) && if(iter == iter2);
6      call(void computeScan_f3(CtrlInfo*, CubeIterator*)) && args(ctrl3,iter3)
7       && if(ctrl == ctrl3) && if(iter == iter3)
8        then executeOldAndNewChain(ctrl,iter);
9      call(void computeScan_f6(CtrlInfo*, CubeIterator*))
10       && args(ctrl6,iter6) && if(ctrl == ctrl6) && if(iter == iter2)
11         then combineDataAndExecutef6(ctrl6,iter6); )
12
13 void executeOldAndNewChain(CtrlInfo* ctrl, CubeIterator* iter) {
14     ctrlNew = ctrl.clone(); iterNew = iter.clone();
15     executeNewChainf3'tof5'(ctrlNew,iterNew);
16     computeScan_f3(ctrl,iter); // execute original chain
17 }
18
19 void combineDataAndExecutef6(CtrlInfo* ctrl, CubeIterator* iter) {
20     combine(); // combines the data (ctrlNew,ctrl,iterNew,iter) and
21                //  stores result in ctrl and iter
22     computeScan_f6(ctrl,iter); // execute last functor with
23                                //  combined data
24 }
```

Listing 3: Sequence aspect for adjoining a chain

```
1  seq(call(void computeScan_f1(CtrlInfo*, CubeIterator*)) && args(ctrl,iter);
2      call(void computeScan_f2(CtrlInfo*, CubeIterator*)) && args(ctrl2,iter2)
3       && if(ctrl == ctrl2) && if(iter == iter2);
4      call(void computeScan_f3(CtrlInfo*, CubeIterator*)) && args(ctrl3,iter3)
5       && if(ctrl == ctrl3) && if(iter == iter3)
6        then replace(ctrl3,iter); )
7
8  void replace(CtrlInfo* ctrl, CubeIterator* iter) {
9      executeNewChainf3'tof5'(ctrl,iter);
10     // computeScanf5' will call computeScanf6 that
11     // then proceeds as usual
12 }
```

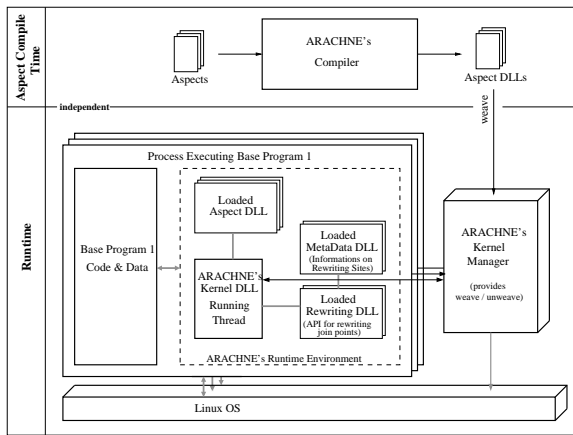**Listing 4: Sequence Aspect for replacing a subchain**

Figure 5: Arachne's Architecture

process, waiting on weaving and unweaving requests. By using a thread for processing the requests and weaving/unweaving, the execution of the base program does not have to be suspended. Upon reception of a weaving request, the Arachne kernel loads the corresponding aspect DLL. Then the kernel loads the rewriting DLLs required by the aspect if necessary. A *rewriting DLL* serves as an API that provides functions to rewrite/instrument one kind of join point. A rewriting DLL does not contain the information about the places to be instrumented in the binary code of the base program, *i.e.* the shadows. This information is stored in the meta data DLLs. *Metadata DLLs* are used to ensure the independence between Arachne's aspect system and the base program. They contain a mapping between the symbolic description of rewriting sites and the actual rewriting sites in the binary code of the base program. To rewrite join points, the rewriting DLL will request the Arachne kernel to load its corresponding metadata DLL. If this metadata DLL has not yet been loaded, the kernel will attempt to generate it by parsing the symbol tables and the object code contained in the base program executable file. Once the corresponding metadata DLL is loaded, the rewriting DLL will use the metadata to resolve the symbolic references in the aspect DLL and the binary code of the base program will be rewritten. For an unweaving request the kernel will instruct the rewriting DLLs referenced by the aspect to restore the original code before unloading the aspect DLL.

5.1.2 Arachne's kernel manager

Arachne's kernel manager serves as an intermediate between the user that wants to weave and unweave aspects into different applications, and aspect kernels that actually weave and unweave the aspects into a specific base program. The kernel manager is embodied in a set of shell commands for the weaving and unweaving of aspects. Once a (un)weaving command is issued by a user, the kernel manager communicates through sockets with the kernel of the specified base program to initiate the (un)weaving. In case a user wants to weave an aspect into a running application that does not yet contain an Arachne kernel, the kernel manager will "inject" an Arachne kernel in the application's address space before instructing it to weave the aspect. First, the Arachne kernel manager suspends the base program execution. Then

it uses debugging APIs (`ptrace` [ref POSIX]) to rewrite the first bytes of the memory image of the base program and restarts the execution of the base program at the beginning of its memory image. Upon execution, the rewritten code loads the Arachne kernel DLL into the address space of the base program and sends a signal to the Arachne kernel manager. Upon reception of the signal, Arachne's kernel manager suspends the execution of the base program, restores the first bytes and finally triggers the continuation of the execution at the place where it interrupted the program execution beforehand. From this point on, the base program contains an Arachne kernel loaded in its address space that can handle weaving and unweaving requests. This dynamic "injection" technique does not require a new start of the application but might just lead to a suspension of the base program execution for up to 500ms.

5.1.3 Arachne's compiler

The compiler is a combination of a lexical analyzer written with Flex and a parser and program generator written with Bison. The compiler first translates an aspect written in the aspect language into a C source file that contains advice in executable functions and dynamic predicates, *i.e.* the residue. Once a join point is rewritten by the aspect runtime, it will automatically trigger the execution of the predicates and in case the aspect applies at the join point, run the appropriate function holding the aspect advice. In a second step, the compiler generates a compiled aspect DLL by using a regular C compiler (`gcc`). As the same aspect can be woven into two different base programs, the information about the rewriting sites, *i.e.* the shadows, of an aspect is not available at aspect compile-time, and thus all references to join points in a base program and rewriting DLLs are in a symbolic form. Once woven, an aspect DLL instructs the aspect runtime environment to instrument the base program at the appropriate places.

5.2 Nuts and bolts of binary code weaving

Arachne's dynamic weaving approach raises a number of issues that have to be considered. First, the base program code must remain executable and stay consistent at all times, and second, the resources (memory and sockets) used by the Arachne kernel should not interfere with the base program.

5.2.1 Consistency of the base program execution

Atomic rewriting of a site. As the rewriting is done at runtime without suspension of the base program execution, Arachne has to ensure that the rewriting does not interfere with the execution. A shadow in the binary code of the base program is rewritten by a jump to the residue of the aspect. However, such a jump instruction is too long for replacing it with one of the atomic memory write operations of the x86 processor. Therefore Arachne uses a rewriting strategy that ensures a consistent execution of the base program. It does so by first inserting a self-referencing loop (short enough to be written atomically) at the beginning of the instruction, so that in case the base program wants to execute the code at the rewriting site, it will just loop. (Note that this synchronization mechanism requires minimal execution time and is compatible with all common higher-level synchronization mechanisms.) Then Arachne writes the end of the jump in-

struction behind the loop before finally rewriting the loop instruction with the beginning of the jump instruction.

Rewriting consistency. In case a join point is composed of a set of assembly instructions, there might be jumps in the base program to an instruction in the set. To conserve the consistency of these jumps, a weaver may not rewrite the whole set of instructions as is done by Kerninst and Dyninst [7, 17, 3], or it might change the jump addresses beforehand. But latter one is nearly impossible to achieve at runtime where jump targets may be determined by an address held in a register or in memory. Therefore Arachne ensures to instrument only the first instruction of the assembly instructions belonging to the join point. At the same time, the instruction that will be rewritten by Arachne has to be big enough to fit a jump instruction. This however is ensured by the selection of the join points that Arachne provides.

Atomic weaving and unweaving. Arachne treats an aspect DLL as a collection of related aspects, potentially collaborating. To ensure a consistent execution of the base program, all rewriting sites addressed by one aspect DLL have to be woven atomically (the same counts for unweaving). This atomicity is provided by executing a dynamic check on the progress of the weaving before the residue and eventually the advice of an aspect are called. Thus only in case all addressed sites are rewritten, the aspect code is executed.

5.2.2 Resource consumption

Rewriting the binary code of the base program at runtime requires the Arachne kernel to share its memory with the base program. To allocate the memory for Arachne, the Linux function `mmap` is called within the base program execution. This function associates a portion of the caller address space with physical memory and treats it as freshly allocated. Since the base program does not know of the enlargement of its address space, it can not interfere with the memory used by the Arachne kernel[1]. In addition, to isolate the socket used by Arachne from the base program and thus avoiding interference between these two, the kernel thread is created through the Linux specific function `clone`, so that only the memory is shared but not the used sockets.

5.3 Extension of Arachne to C++

We conclude this implementation section by considering the design of how to extend Arachne for dynamic weaving of C++ applications, such that Siemens AG's code base for medical image generation can be used without previously mapping it to C.

Essentially, C++ is a typed object-oriented extension of C providing function overloading and overriding, instance variables and compile-time code generation facilities (*i.e.* `template`). To ensure proper interoperability between compilers, the compiled representation of a C++ file has been normalized [5, 15]. Except from the language features specific to C++, this standard closely follows the ANSI C spec-

---

[1]Contrary to `mmap`, regular memory allocators typically have some sort of side effect. For example, in case the Arachne kernel was built with the GNU `malloc` command, the base program could have used `sbrk(0)` to detect the memory allocations performed by the Arachne kernel.

ification. Therefore, the techniques used in Arachne to instrument C programs are directly applicable to C++ programs and only the features specific to C++ require further considerations and will be discussed in the following.

C++ implements function overloading by encoding the types of the signature in the function name. This encoding process is defined by the standard and allows tools such as `GNU nm` to retrieve the exact, source level name from the encoded, binary level function name. By using this property, Arachne will be able to properly handle overloaded functions.

Function overriding in C++ is implemented using `vtable`s [5, 15]. The C++ compiler translates the invocation of `virtual` functions into binary code that will first retrieve the address to the function to be executed from the `vtable` before actually executing it. In addition the C++ compiler holds each `vtable` as a global variable. Therefore, to trigger the execution of an action upon a virtual function, Arachne can just replace the addresses stored in the `vtable` by the address of the action.

Because of the standardization of the memory layout of object instances [5, 15], the techniques used by Arachne to track global and local C variables can easily be adapted to cover instance variables.

Finally, C++ compile-time generation facilities will not interfere with Arachne for C++. Arachne will however not be able to trigger advices on the metacomputation performed by `template` functions, since these computations are performed at compile-time and their results are inlined in the compiled executable. For the computations performed at runtime, *i.e.* template mechanisms used to parameterize a class or function, the necessary information is encoded within the binary names of the functions and variables and may be used by Arachne.

6. RELATED WORK

Image generation by medical devices is an active research field [12, 2, 18, 14]. Despite rapid evolutions, industrial medical software offers a fixed, closed set of features (functors). Hence, the state of the functor graph required for each image processing could be fixed at compile-time. However, the combinatorial explosion makes this approach unsuitable without appropriate tools.

Partial evaluation systems could be used to master such an combinatorial explosion. To be successful, such an approach would require a design where all image treatments will be derived from a most generic one. Ideally, in a partial evaluation approach, the application should use only a single functor graph capable of performing any image processing. Partial evaluation techniques and tools [8, 13] could then be used to automatically prune the unused functors from the functor graph depending on its use in the different parts of the program. But this generic and complete graph does not exist for Siemens AG's medical devices and tools for partial evaluation are rather unwieldy compared to, *e.g.*, Arachne.

To our knowledge, Arachne is the only dynamic aspect weaving system for C. AspectC [4] (for which no tool support is available) and AspectC++ [16] extend C and C++, respectively, by an aspect model very similar to AspectJ's [9]. Both of these provide static weaving and therefore do not meet Siemens AG's requirements of dynamic adaptability. Furthermore static approaches would require the implementation of sophisticated (and probably complex) undo-

mechanisms to support a notion of reversibility, similar to that built-in into Arachne.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented part of the existing code base of Siemens AG, Germany, for the generation of images by medical devices. We have presented some interactive adaptation scenarios arising in practice. We have also motivated that the corresponding transformation of the structure of image generation algorithms should benefit from AOP techniques. We have then outlined a solution realizing the adaptation scenarios in form of aspects using the aspect language of Arachne, a dynamic weaver for C programs. Finally, binary code weaving for C has been detailed and the design of an extension of Arachne for C++ has been presented.

There are several direct leads to pursue the work presented in this paper. Most prominently, the set of adaptation scenarios should be completed. Second, an implementation of the C++-version and the corresponding transformations is to be done. Finally, the relation between partial evaluation techniques and our AOP-based approach should be investigated.

8. REFERENCES

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an os kernel using temporal logic and aop. In *Proceedings of Automated Software Engineering (ASE'03)*, pages 196–204. IEEE, 2003.

[2] J. Ashburner and K.J. Friston. Why voxel-based morphometry should be used. *NeuroImage*, 14(6):1238–1243, 2001.

[3] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[4] Y. Coady, G. Kiczales, J.S. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can aspect-oriented programming help? In *Proceedings of the Tenth ACM SIGOPS European Workshop*, pages 79–86, St. Emilion, France, September 2002.

[5] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SG, editors. *Itanium C++ ABI*. CodeSourcery, November 2003. published on-line http://www.codesourcery.com/cxx-abi/abi.html.

[6] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *Proc. of 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, March 2005. To appear.

[7] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, and L. Zheng. MDL: A language and compiler for dynamic program instrumentation. In *IEEE Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 201–213, November 1997.

[8] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, Sep. 1996.

[9] G. Kiczales et al. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin, June 2001.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, et al. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *LNCS*, pages 327–353. Springer Verlag, Berlin, June 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, et al. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[12] G. Lohmann, K. Muller, V. Bosch, H. Mentzel, S. Hessler, L. Chen, S. Zysset, and D.Y. von Cramon. Lipsia a new software system for the evaluation of functional magnetic resonance images of the human brain. *Computerized Medical Imaging and Graphics*, 25(6):449–457, 2001.

[13] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, May 2001.

[14] W.D. Penny, N. Trujillo-Bareto, and K.J. Friston. Bayesian fMRI time series analysis with spatial priors. *NeuroImage*, 2004. Accepted.

[15] Nathan Sidwell. A common vendor c++ abi. In *Proceedings of the Association of the C and C++ Users conference*, April 2003.

[16] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.

[17] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[18] D. Veltman, A. Mechelli, K.J. Friston, and C.J. Price. The importance of distributed sampling in blocked functional magnetic resonance imaging designs. *NeuroImage*, 17(3):1203–1206, 2002.