

Automating adaptive image generation for medical devices using Aspect-Oriented Programming

Thomas Fritz^{a,1}, Marc Ségura^b, Mario Südholt^b, Egon Wuchner^c, Jean-Marc Menaud^b

^aInstitut für Informatik, Gruppe PST
Ludwig-Maximilians-Universität
Oettingenstraße 67
80538 München, Deutschland
fritzt@informatik.uni-muenchen.de

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

^cCorporate Technology, SE2
Siemens AG
Otto-Hahn-Ring 6
81739 München, Deutschland
Egon.Wuchner@siemens.com

Abstract

Image generation, e.g., in computer tomographs, requires the use of sophisticated algorithms which 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, e.g., to individual patients. On the application level, such characteristics are frequently scattered all over the code of the application. This suggests the use of Aspect-Oriented Programming (AOP) techniques to modularize such crosscutting functionality.

In this paper we present an approach to automate image generation tasks using AOP and their application in the context of medical devices from Siemens AG, Germany. Concretely, we present three results: (i) a motivation why imaging software can benefit from dynamic AOP, (ii) a case study of how image generation, in particular for medical devices, can be adapted using the Arachne system for dynamic AOP in C, and (iii) a suitable aspect language and its realization within Arachne.

1 Motivation: image generation and AOP

Image generation algorithms, e.g., in the medical sector, frequently map measured signals into some form amenable for interpretation by humans (doctors) using sophisticated compositions of a large number of basic image manipulation operators. For instance, many medical devices, such as 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).

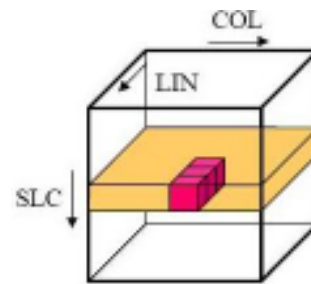


Figure 1. Space cube of measured signals associated with a human body part

The signal specific to a particular position within the cube is characterized by its wave periodicity, 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 as well as the corresponding part of the examined human body. 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 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 of basic functors, 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

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

of a set of complete functor sequences generating an image for a patient, followed by other complete sequences, if necessary, for the same patient. From a usability point of view, this means that doctors can only start one of several functor sequences which have been predefined at system construction time. Furthermore, medical personnel can parametrize functor sequences with values. This adaptation is very limited, though, and does not allow, e.g., to reorganize functor sequences.

However, following customer requests, an evaluation is performed within Siemens medical devices unit of explicit support for dynamic and automatic 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 is especially useful to optimize the final images *w.r.t.* individual patients. 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 image generation, code is executed corresponding to sequential and parallel functor sequences, the latter implementing, e.g., calculations of different parts using different resolutions. Execution can be represented by another graph, henceforth called the functor graph, a sub-graph of the graph of all valid transformations introduced 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.

Scattered functionalities, which cannot reasonably be modularized using traditional programming means (such as object-oriented programming), is the subject matter of Aspect-Oriented Programming [10], an emerging part of the software engineering domain. AOP focuses on language mechanisms for so-called aspects, which enable the modularization of such functionalities. AOP also investigates corresponding implementation support, so that aspects can be added to existing applications. This is particularly useful to adapt applications by new functionalities and AOP has been applied to 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 automation of the adaptation of such image generation software seems therefore promising. In particular, an AO approach realizing this adaptation problem through (relatively) small changes to an ex-

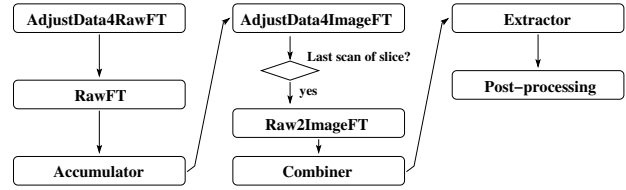


Figure 2. Basic steps for image generation

isting 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 several results of how to address the adaptation problems for image generation software and show how the corresponding techniques can be applied to 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. We also present a suitable aspect language featuring a sequence aspect construct and give an overview of Arachne’s implementation as well as its on-going extension to C++. This work extends our previous work [7] by a more detailed description of the underlying application domain, a detailed presentation of sequence aspects (especially their implementation) and a performance evaluation.

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.

2 Imaging code base

In this section we give an overview of the C++ code base used for image generation in Siemens devices and present some typical code patterns used for image generation tasks.

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) 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.

Such image generation functors 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 multi-dimensional 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 takes a cube as input, calculates some values and has to store the results. This is done either ‘in-place’ by overwriting certain parts of the same input cube, by retrieving a global cube or creating a new one. As an example `Raw2ImageFT` takes a raw data cube to retrieve input data and creates an `ImageCube` in order to store constructed images in a separate cube of images.

Compared to this second Fourier transformation the first one, `RawFT`, works in-place on the same raw data cube (see List. 1, lines 7, 10–27). Passing the cube to `computeScan` is done implicitly by the `CubeIterator` argument. Such iterators require a cube for their correct construction and allow to iterate over a cube with respect to the set of dimensions the respective generation step is interested in.

As an example, `computeScan` of functor `RawFt` initializes a local iterator instance by copying the iterator argument (see the iterator `rawFtIter` in lines 20–23 of List. 1). It prepares the cube as input to the underlying Fourier transformation by specifying the current line and slice as constants. In addition it specifies an iteration range covering columns and receiver channels. The underlying Fourier transformation `Imager::FT` takes the prepared iterator, calculates intermediary values and stores them in-place in the same cube according to the column and channel indices of the iterator range.

Functors inherit from a base class `Functor` as

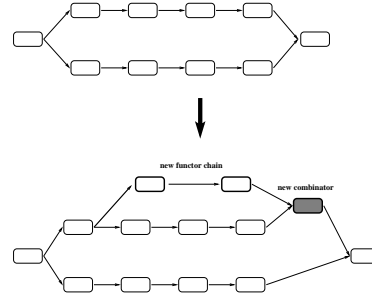


Figure 3. Adding a functor chain in parallel

shown for `RawFT`. 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. 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 control of automatically applied adaptations. Technically, these adaptations take the form of transformations of the graph defining the functor sequences which generate images from raw data.

In a first scenario, a doctor using a tomograph discovers some indication of a tumor during an on-going scan. Thus, he decides to examine the corresponding region in more detail without losing the currently calculated image information and without modification of the image generation process of the other image parts. This can be done by augmenting the initial image generation sequence by a new sequence of functors performing a very detailed image calculation for the body section to be focused on. Technically, this scenario requires the *adjunction of a new parallel functor chain* to an existing chain of the current functor graph, as illustrated by Fig. 3. This figure shows the typical application of a transformation to a graph consisting of two parallel functor sequences. Both chains are then executed in a pseudo-parallel fashion and the resulting images of both functor chains are finally combined to one image per slice.

A second scenario consists in tomography sessions during which some generated part of an image has to be rendered differently, *e.g.*, using a false color representation. Technically, this scenario requires the *replacement of a part of a functor chain* by another one as shown in Fig. 4.

To conclude the discussion of image processing adaptations, note that functors affect several connection points of the original chain. Furthermore, many adaptations are performed during a tomography examination. Adaptations may be applied to the initial functor chain but also

```

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 void RawFT::computeScan( CtrlInfo& ctrl, CubeIterator& iter ) {
10     CubeIterator rawFtIter( iter ); // copy input iterator
11
12     // set the cube dimensions and ranges the RawFT should work on
13     rawFtIter.init( COLUMN, 0, ctrl.getNumberOfColumns(),
14                   LINE, ctrl.getCurrentLine(), ctrl.getCurrentLine(),
15                   SLICE, ctrl.getCurrentSlice(), ctrl.getCurrentSlice(),
16                   CHANNEL, 0, ctrl.getNumberOfChannels());
17
18     Imager::FT( iter, rawFtIter ); // call RawFT
19
20     for(int i, i<FunctorList.size(), i++){ // call next Functors
21         this->getNextFunctor(i)->computeScan( ctrl, iter ); } };

```

Listing 1. RawFT implementation skeleton

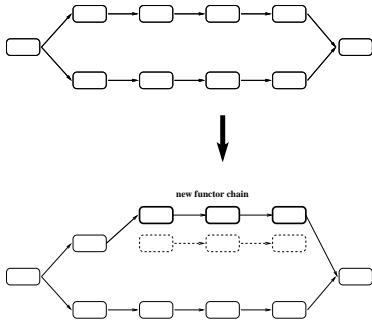


Figure 4. Replacing a functor chain

to chains which have been dynamically added previously. Hence, the corresponding transformations are spatially and temporally scattered over the entire functor graphs.

4 Applying dynamic AOP

We now turn to the problem how to express the adaptation scenarios using the Arachne system for Aspect-Oriented Programming of applications developed using the C language. (Note that since Arachne-C++ is in the final phases of development we have performed the experiments reported here after transformation of the functors from C++ to C; the transformation has been quite simple because the functors, cf. the example in the previous section, do not make extensive use of the object-oriented features of C++.)

Before introducing Arachne’s aspect language, which we use for the automation of adaptive image generation, let us introduce the relevant basic notions of Aspect-Oriented Programming. AOP languages define aspects using two main abstractions: “pointcuts” which define where a base application has to be modified by an aspect and “advice” which defines what modifications are to be

applied at execution points matched by pointcuts. In the context of image generation based on functor sequences, pointcuts identify points within such sequences and advice corresponds to a transformation of functor sequences.

Arachne’s aspect language. Arachne’s aspect language (see [6] for a more detailed presentation) provides analogues for C to AspectJ’s main Java-oriented 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., for variable access, see [6].)

Arachne supports the concise formulation of aspects over functor sequences through a feature, which distinguishes it from most other aspect languages (including AspectJ): a construct for explicit sequencing on the language level. The sequence-construct is of the following form:

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

where *Prim* is a primitive aspect, such as

```
call(void m(int,long)) then mNew();
```

A primitive aspect associates a pointcut to an advice. The former matches, e.g., a call, the latter typically is a function call which is to be executed instead of the call matched by the pointcut 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 repetition operation ‘*’ is used), i.e., a primitive aspect in the sequence has priority over its predecessor. Overall, the sequence construct

```

1 CtrlInfo* ctrlNew; CubeIterator* iterNew;
2 seq(call(void computeScan_f1(CtrlInfo*, CubeIterator*)) && args(ctrl1,iter);
3   call(void computeScan_f2(CtrlInfo*, CubeIterator*)) && args(ctrl2,iter2)
4     && if(ctrl == ctrl2) && if(iter == iter2);
5   call(void computeScan_f3(CtrlInfo*, CubeIterator*)) && args(ctrl3,iter3)
6     && if(ctrl == ctrl3) && if(iter == iter3)
7     then executeOldAndNewChain(ctrl1,iter);
8   call(void computeScan_f6(CtrlInfo*, CubeIterator*))
9     && args(ctrl6,iter6) && if(ctrl == ctrl6) && if(iter == iter2)
10    then combineDataAndExecutef6(ctrl6,iter6); )
11 void executeOldAndNewChain(CtrlInfo* ctrl, CubeIterator* iter) {
12   ctrlNew = ctrl.clone(); iterNew = iter.clone();
13   executeNewChainf3Newtof4New(ctrlNew,iterNew);
14   computeScan_f3(ctrl,iter); } // execute original chain
15 void combineDataAndExecutef6(CtrlInfo* ctrl, CubeIterator* iter) {
16   combine(); // combines data from two chains and store result in (ctrl, iter)
17   computeScan_f6(ctrl,iter); } // execute last functor with combined data

```

Listing 2. Sequence aspect for adjoining a chain

```

1 seq(call(void computeScan_f1(CtrlInfo*, CubeIterator*)) && args(ctrl1,iter);
2   call(void computeScan_f2(CtrlInfo*, CubeIterator*)) && args(ctrl2,iter2) && if(ctrl==ctrl2)
3     && if(iter==iter2);
4   call(void computeScan_f3(CtrlInfo*, CubeIterator*)) && args(ctrl3,iter3) && if(ctrl==ctrl3)
5     && if(iter==iter3) then replace(ctrl3,iter);)
6 void replace(CtrlInfo* ctrl, CubeIterator* iter){
7   executeNewChainf3Newtof5New(ctrl,iter); //computeScanf6 then proceed as usual
8 }

```

Listing 3. Sequence Aspect for replacing a subchain

is best understood as a means to relate different execution events over time (as well as corresponding advice).

In addition, within a sequence the language allows some information to be made explicit about the execution event matching some primitive aspect. Programmers can, e.g., retrieve the arguments associated with a function invocation using the constructor `args` (another constructor permits to match return values) as exemplified by:

```

call(void m(int,long)) && args(v1,v2)
then mNew(v1,v2);

```

The information collected as part of a sequence is discarded upon a match of the last primitive aspect in the sequence. Furthermore, Arachne includes an `if` keyword enabling aspect programmers to test conditions as part of primitive aspects as shown in the following:

```

call(void m(int,long)) && args(v1,v2)
&& if(v1==0) then mNew(v1,v2);

```

Note that by means of argument matching (and similar constructors) within a sequences and corresponding conditionals, a sequence construct can be seen as defining data-flow relationships between primitive aspects.

Adjoining a new functor sequence. A new functor sequence can be adjoining 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 (`f3New` to `f4New`)

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. 2.

The `computeScan` functions of `f1` and `f2` are executed as usual, but once `f3` is reached (lines 6–8 in List 2) 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 `f3New` to `f5New` (cf. Fig. 4). The aspect in List. 3 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 `f5New` 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

In this section we describe how Arachne enables dynamic weaving and unweaving of aspects into running legacy C applications. After introducing Arachne’s architecture, this section focuses on the implementation of the `seq` construct. Furthermore, because of the encouraging results reported in the previous sections, we extended Arachne to deal with C++ applications. Such an extension is important: Siemens AG’s code base is the result of 8 years of work and is of considerable size. Hence, the ability to reuse it *as is* is not a matter of convenience but a feasibility criterion. By targeting Arachne directly to C++ code we obviate the need for the mapping to C introduced in the preceding sections. Finally we present performance evaluations of our system.

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. The aspect compiler translates pointcuts into C code that is later compiled with `gcc`. The runtime environment manages the joinpoints themselves: it rewrites the compiled code of the base program interpreted by the processor such that pointcuts are matched at appropriate joinpoints.

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. Once the kernel DLL is loaded in the address space, it creates a thread in the base program process, waiting for weaving and unweaving requests. By using a thread for processing the weaving and unweaving requests, 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. *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. In case of unweaving, the kernel instructs the rewriting DLLs referenced by the aspect to restore the original code before unloading the aspect DLL.

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 specific base programs.

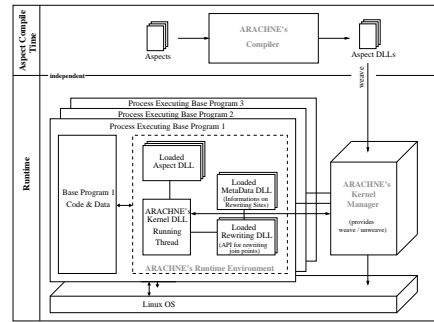


Figure 5. Arachne’s Architecture

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, which test whether a pointcut matches during execution. In a second step, the compiler generates a compiled aspect DLL by using a regular C compiler (`gcc`).

5.2 `seq` implementation

In Arachne, the `seq` construct defines regular sequences of primitive aspects. If the pointcut of a primitive aspect matches, the aspect environment runs the appropriate advice. Each primitive aspect can collect information, such as argument and return values, from the base program execution: this is useful, *e.g.*, to pass the appropriate arguments while replacing a functor by another one. In the following, we describe how Arachne implements state management during execution of a sequence and explain how the runtime tracks progress in a sequence.

The aspect compiler translates the sequence aspect into a finite state automaton. Each time the pointcut of the first primitive aspect of the sequence is matched, a new sequence starts in the first state of the automaton. At such points, Arachne’s runtime environment allocates a structure, *i.e.*, a sequence instance, acting as a container for the information collected during the lifetime of the sequence. Each step in the sequence is associated with one state of the finite automaton of the sequence and a linked list holding sequence instances currently in that state. When an event in the execution of the base program matches the static part of a pointcut of a step in the sequence, the dynamic part, *i.e.*, the part dependent on dynamic conditions on the data, is checked for every instance currently in that state. For each sequence instance for which the dynamic condition holds the appropriate advice is run.

In case a sequence step contains a repetition (`*`) and an event matches the pointcut of the following step in the sequence, the dynamic check is also performed on the instances of the preceding step. If the dynamic condition holds for such an instance, the advice is executed and the instance is advanced to the next step, otherwise it is left in its original linked list. After the execution of the advice associated to the last step in the sequence, the memory allocated for the instance, for which that advice has been

executed, is freed.

5.3 Extension of Arachne to C++

In order to forego the need for a translation of functors from C++ to C, we now consider how we extended Arachne for dynamic weaving into C++ applications.

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, 14]. Except from the language features specific to C++, this standard closely follows the ANSI C specification. 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 properly handles overloaded functions.

Function overriding in C++ is implemented using vtables [5, 14]. The C++ compiler translates the invocation of virtual functions into binary code that first retrieves 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 replace the addresses stored in the vtable by the address of the advice.

Finally, C++ compile-time generation facilities do not interfere with Arachne for C++. Arachne however is not able to trigger advice on template functions, since these computations are performed at compile-time and their results are inlined in the compiled executable. This is not surprising as Arachne for C is not able to trigger advice upon inlined C functions.

5.4 Performance evaluation

To evaluate the cost for each construct of our aspect language, we performed a number of microbenchmarks comparing C code constructs with the application of (empty) Arachne aspects that achieve the same effect.

Figure 6 summarizes our results. Using the aspect language to replace a function that returns immediately is only 1.3 times slower than a direct, aspect-less, call to that empty function (28 to 21 cycles). This good result is not surprising : the aspect compiler rewrites advices as regular C functions, and also a call pointcut involves no overhead. A seq of n invocations of such empty functions is approximately n times slower than n direct, aspect-less, successive function calls. Compared to the pointcuts used to delimit the different stages, the seq overhead is limited to a few pointer exchanges between the linked

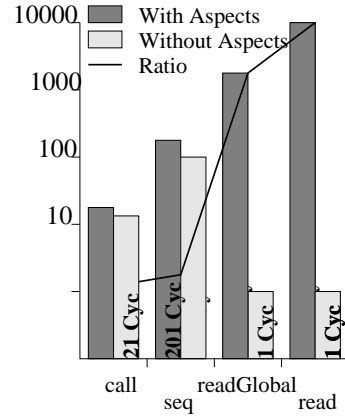


Figure 6. Pointcut Performance

lists holding the bound variable. Without aspects, a direct global variable read is usually carried out in a single unique x86 processor cycle. Upon a global variable access, the Arachne runtime has to save and restore the processor state to ensure the execution coherency, as advices are packaged as regular C functions. Hence a global variable readGlobal is about 2700 times slower than a direct, aspect-less global variable read. read performance can be explained in the same way: in the absence of aspect, local variables are accessed in a single unique x86 processor cycle. seq and controlflow can involve several points in the execution of the base program (*i.e.* different stages for seq and different function invocations for the controlflow). The runtime of these pointcuts increases linearly with the number of execution points they refer to and with the number of matching instances.

While some of the above aspect operations weigh heavily in microbenchmarks, as shown in [6], the overhead of typical uses of Arachne aspects are negligible, *i.e.*, completely amortized, in real-world applications. This result should also hold in the present image manipulation application, because basic image manipulation functors, which are not modified by aspect, encapsulate complex and hence costly calculations. Operations introduced via aspects, however, occur infrequently.

6 Related work

Image generation by medical devices is an active research field [11, 3, 16, 13]. 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 a 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, 12] 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++ [15] 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. DAC++ [2] is a recent dynamic aspect weaver for C++. However, this work is geared towards optimization of weaving in multi-threaded environments, which is a minor problem in our application context. Furthermore, no corresponding tool is available.

7 Conclusion and future work

The quality of the images generated by medical devices is crucial for physicians. Yet the complexity and lack of flexibility of existing image generation code makes it difficult to tailor the image generation process to individual patients. Technically, this is a consequence of the necessary adaptations involving changes scattered all over imaging software. In this paper, we presented part of the existing code base used by Siemens AG, Germany, for the generation of images by tomography devices. We have shown how an aspect-oriented approach allowing on-the-fly automated adaptations of the image generation process based on transformations of functor sequences. We have motivated different real-world scenarios requiring such transformations and have shown how to realize them using the aspect language of the Arachne, a dynamic aspect system originally devised for C applications. Finally, we have presented an extension of Arachne to C++ in order to be able to manipulate Siemens AG's code base as is.

In the future we intend to broaden or shorten the scope of the adaptation scenarios embedded within the tomograph. In addition we plan to investigate how to facilitate the design of adaptation scenarios can be improved by C++ support for aspect language.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *SE'03*, pages 196–204, 2003.

[2] S. Almajali and T. Elrad. Coupling availability and efficiency for aspect-oriented runtime weaving sys-

tems. Proc. of the Dynamic Aspects Workshop (DAW'05) at AOSD, Mar. 2005.

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

[4] Y. Coady, G. Kiczales, J. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can Aspect-Oriented Programming help? In *Proc. of the Tenth ACM SIGOPS European Workshop*, pages 79–86, Sept. 2002.

[5] CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SG, editors. *Itanium C++ ABI*. CodeSourcery, Nov. 2003.

[6] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with Arachne. In *AOSD'05*. ACM Press, Mar. 2005.

[7] T. Fritz, M. Ségura, M. Südholt, E. Wuchner, and J.-M. Menaud. An application of dynamic AOP to medical image generation. Dynamic Aspects Workshop (DAW'05) at AOSD, Mar. 2005.

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

[9] G. Kiczales, E. Hilsdale, J. Hugunin, et al. An overview of AspectJ. In *ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer Verlag, Berlin, June 2001.

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

[11] G. Lohmann, K. Muller, V. Bosch, H. Mentzel, S. Hessler, L. Chen, S. Zysset, and D. 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.

[12] 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.

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

[14] N. Sidwell. A common vendor C++ ABI. In *Proc. of the Association of the C and C++ Users conference*, Apr. 2003.

[15] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *TOOLS Pacific 2002*, Sydney, Australia, Feb. 2002.

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