Cardoso et al. EURASIP Journal on Embedded Systems 2012, **2012**:7
http://jes.eurasipjournals.com/content/2012/1/7

EURASIP Journal on Embedded Systems
a SpringerOpen Journal

RESEARCH Open Access

A generative-oriented model-driven design environment for customizable video surveillance systems

Nuno Cardoso[1*], Pedro Rodrigues[1], João Vale[1], Paulo Garcia[1], Paulo Cardoso[1], João Monteiro[1], Jorge Cabral[1], José Mendes[1], Mongkol Ekpanyapong[2] and Adriano Tavares[1]

Abstract

To tackle the growing complexity and huge demand for tailored domestic video surveillance systems along with a high demanding time-to-market expectation, engineers at IVV Automação, LDA[a] are exploiting video surveillance domain as families of systems that can be developed following a pay-as-you-go fashion rather than developing an ex-nihilo new product. Several and different new functionalities are required for each new product's hardware platforms (e.g., ranging from mobile phone, PDA to desktop PC) and operating systems (e.g., flavors of Linux, Windows and MAC OS X). Some of these functionalities have special economical constraints of development time and memory footprint. To better accommodate all the above listing requirements, a model-driven generative software development paradigm supported by mainstream tools is proposed to offer a significant leverage in hiding commonalities and configuring variabilities across families of video surveillance products while maintaining the new product quality.

1 Introduction

Nowadays there is a growing demand for video surveillance systems [1]. The development of these systems is increasingly complex since there is a high demand for new products with a rising number of different requirements [2].

However, it can be noticed that the increasing complexity is induced by the variability in tasks related to image capturing, image processing, communications and computer vision [3]. Furthermore, it is now expected that the system runs in different hardware platforms, ranging from desktop PCs to low cost embedded boards, mobile phones, etc.

The majority of system designers use modular architectures based on Microsoft DirectShow implementation [4,5] in order to address the growing complexity of novel products and the ever increasing time-to-market pressure. In this implementation, all system functionalities are implemented as individual plug-ins or filters. Later, these plug-ins will be connected following the pipeline execution model (filter graph) through a generic interface. In this way, higher flexibility, customizability and reusability can be achieved for video surveillance oriented design. Furthermore, at run time, only the required plug-ins for a given configuration will be loaded and a new system configuration can be implemented by simply defining new plug-ins to be loaded and the connections among them.

In the case of embedded systems, such an implementation incurs a huge abstraction penalty [6-9], since it is based on weighty language's dialects (i.e., features that increase overhead in space and performance), like dynamic polymorphism.

In order to tackle these embedded systems constraints [6], video surveillance systems can be implemented as a software product line (SPL) [2,10] by designing plug-ins which fit all possible video surveillance system's configurations and simply reusing the common features of several configurations while considering the variable features that identify each system specific configuration.

Based on practical experience at IVV in tackling the design challenges of video surveillance systems, new software technologies such as model driven development [11] and generative programming [12], should be

*Correspondence: linun77@gmail.com
[1]Department of Industrial Electronics, Centro Algoritmi, University of Minho, Braga, Portugal
Full list of author information is available at the end of the article

Springer

© 2012 Cardoso et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

combined with SPL engineering to bridge the abstraction gap between domain modeling and feature modeling.

The remainder of this article is structured as follows. The following section deliberates related works. Next, in Section 3, Microsoft DirectShow is reviewed, together with brief discussion on how to reverse engineer it. Section 4 focuses on video surveillance domain engineering by implementing the feature modeling. Section 5 presents the details of video surveillance application engineering, showing how model-driven engineering techniques are employed to establish the relationships between video surveillance feature models and video surveillance domain model. It also describes how offline and online partial evaluators [13] will be applied, respectively, at whole system and filter designs. In Section 6, the implementation of a SDK for video surveillance product line development is described. Section 7 presents some results. Finally, in Section 8 we conclude the article and present some directions of future study.

2 Related work

The amount of research works in this field is very large, so this short literature survey describes related works that apply similar software technologies, describes video surveillance systems design and/or manages variability.

In [2], a mixture of model-driven and generative programming techniques are proposed to support variability configuration of video surveillance systems. In order to better manage features combination at runtime, it separates the variability in domain and code representation spaces.

In [14], Aspect-Oriented Programming (AOP) is employed in order to avoid crosscutting concerns by managing the existing variability in operating systems domain. Even though the use of AOP enables higher levels of granularity when compared with other variability management strategies, it incurs in higher overhead depending on the applied type of advice and the inability to manage variability in systems with several instances of the same class each with different requirements.

In [15], to better tackle the time-to-market pressure during the developing of digital games the use of generative programming was proposed. In [16], a Fujaba plug-in for SPL development was described using model-driven techniques. The main focus is on how to bridge the gap between feature modeling and domain modeling with annotations.

In [17], new mechanisms are explored in order to better specify and understand the relationship between system variability and requirements by combining SPL, Aspect-orientation and model-driven techniques. The Ample project, presents two complementary languages, RDL and VML4RE, to allow specifying and reasoning about crosscutting relationships between features and requirements.

In [18], Matilda is described. Matilda is a model-driven development framework that allows automatic code generation through transformations using UML models while addresses the abstraction gap and lack of traceability related to current model-driven development practice. In this way, the developer analyses, designs and executes applications in modeling layer, abstracting it from the programming layer.

In [19], generative programming as an automatic selection and assembly of components based on configuration knowledge and a set of construction rules is proposed. A layered architecture, called GenVoca is used and the configuration knowledge is implemented using C++ template metaprogramming as generator.

3 Microsoft directshow and its reverse engineering

DirectShow is a pipeline-based media-streaming architecture and API based on component object model (COM) framework that provides a common interface for media across several programming languages. It's a filter-based framework that implements a complex multimedia application by connecting several filters of three main kinds (i.e., source, transform and renderer filters) through their input and output pins using filters graphs to provide specific functionality [5].

Since DirectShow SDK only offers partial code of the framework, we run a reverse engineering process using IDA Pro [20] on *quartz.dll* to figure out its real architecture, starting with the *CFilterGraph* class.

Due to the limited space, we'll only present how the implemented interfaces at *CFilterGraph* class were identified as well as the overhead of a filter class like *CMjpegDec*. Listing 1 partially shows the assembly code of the *CFilterGraph* constructor where the initialization of the virtual tables for each defined interface is visible. The register *esi* points to the data member of the class (i.e., *this* pointer) while the first memory addresses point to the interfaces virtual tables implemented by *CFilterGraph* class. Figure 1 presents *CFilterGraph* class layout with special focus on its thirteen direct and ten indirect interfaces through the pointer, *m_pFGControl*, to an object of *CFGControl* class that encapsulates all filter graph control functionalities.

3.1 Listing 1 Partial Assembly code of *CFilterGraph* class constructor

```
1 ??0CFilterGraph@@AAE@PAGPAUIUnknown@@-
  PAJ@Z proc near
2
3 ... mov esi, ecx ; esi = this... call
4 CBaseFilter::CBaseFilter(ushort const *,IUnknown
  *,CCritSec *,_GUID
5 const &) push [ebp+cbData] mov dword ptr
  [ebx], offset
```

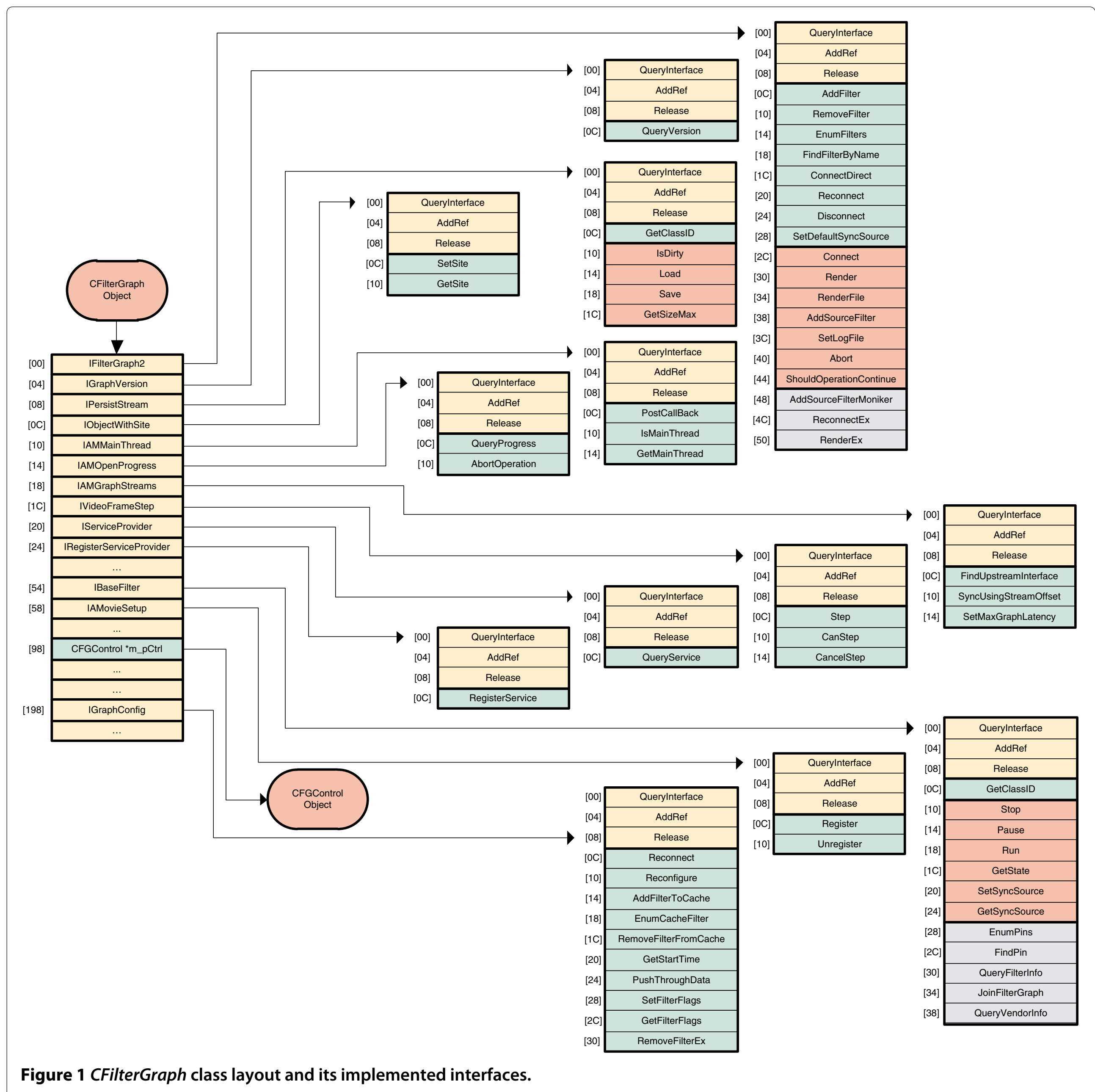

Figure 1 ***CFilterGraph*** **class layout and its implemented interfaces.**

6 CFilterGraph::'vftable'{for 'CUnknown'} **xor ebx, ebx lea ecx,**
7 **[esi**+9**Ch] mov dword ptr [esi], offset**
8 CFilterGraph::'vftable'{for 'IFilterGraph2'} **mov dword ptr**
9 **[esi**+4**]**, **offset** CFilterGraph::'vftable'{for 'IGraphVersion'} **mov**
10 **dword ptr [esi**+8**]**, **offset** CFilterGraph::'vftable'{for
11 'IPersistStream'} **mov dword ptr [esi**+0**Ch]**, **offset**
12 CFilterGraph::'vftable'{for 'IObjectWithSite'} **mov dword ptr**
13 **[esi**+10h**]**, **offset** CFilterGraph::'vftable'{for 'IAMMainThread'} **mov**
14 **dword ptr [esi**+14h**]**, **offset** CFilterGraph::'vftable'{for
15 'IAMOpenProgress'} **mov dword ptr [esi**+18h**]**, **offset**
16 CFilterGraph::'vftable'{for 'IAMGraphStreams'} **mov dword ptr**
17 **[esi**+1**Ch]**, **offset** CFilterGraph::'vftable'{for 'IVideoFrameStep'} **mov**
18 **dword ptr [edi]**, **offset** CFilterGraph::'vftable'{for

19 'IServiceProvider'} **mov dword ptr [esi**+24h], **offset**
20 CFilterGraph::'vftable'{for 'IRegisterServiceProvider'} **mov dword**
21 **ptr [esi**+54h], **offset** CFilterGraph::'vftable'{for 'IBaseFilter'} **mov**
22 **dword ptr [esi**+58h], **offset** CFilterGraph::'vftable'{for
23 'IAMovieSetup'} **mov [esi**+98h], ebx...

From the *CMjpegDec* class constructor analysis, Listing 2, the following code segments were identified, informing us that it inherits from *CVideoTransformFilter* class and implements *IUnknown, IBaseFilter* and *IAMovieSetup* interfaces:

i. Initialization of the stack frame from line 1 to 3;
ii. Calling of *CVideoTransformFilter* constructor as a base class from line 6 to 10;
iii. Registering of virtual tables addresses for each implemented interface from line 15 to 17;
iv. Initialization of class variable members from line 18 to 24;
v. Returning to the caller with stack restoring from line 25 to 31.

3.2 Listing 2 *CMjpegDec* class constructor assembly code

1 **mov edi, edi push ebp** *; save ebp mov ebp, esp ;*
2 save stack frame **push esi** *; save esi push edi ;*
3 save **edi push offset** _CLSID_MjpegDec *; push CLSID_MjpegDec*
4 (REFCLSID) **push [ebp**+arg_4] *; push pUnk (LPUNKNOWN) mov*
5 **esi, ecx** *; esi = this push [ebp+arg_0] ; push pName (TCHAR*
6 *) **call** CVideoTransformFilter::CVideoTransformFilter(...) *; call*
7 the base class **xor edi**, **edi** *; edi = 0 push edi*
8 *; push nDefault (INT) push offset aMjpegnoskip ; push*
9 "MJPEGNoSkip" (LPCTSTR lpKeyName) **push offset** AppName *; push*
10 "Quartz" (LPCTSTR lpAppName) **mov dword ptr [esi]**, **offset**
11 CMjpegDec::'vftable'{for 'CUnknown'} **mov dword ptr [esi**+0**Ch]**,
12 **offset** CMjpegDec::'vftable'{for 'IBaseFilter'} **mov dword ptr**
13 **[esi**+10h], **offset** CMpegAudioCodec::'vftable' for 'IAMovieSetup' **mov**
14 **[esi**+0C0h], **edi mov [esi**+0C8h], **edi mov [esi**+0CCh], **edi mov**
15 **[esi**+0D0h], **edi mov [esi**+0D4h], **edi call**
16 GetProfileIntW(x,x,x) *; GetProfileInt("Quartz", "MJPEGNoSkip", 0);*
17 **mov [esi**+0B0h], **eax mov eax, [ebp**+arg_8] *; eax = phr;*
18 (HRESULT ∗phr) **mov [eax], edi** ; ∗*phr = NOERROR; pop*
19 **edi** *; restore edi mov eax, esi ; ret = this pop*
20 **esi** *; restore esi pop ebp ; restore ebp retn*
21 0**Ch** *; return and clean stack*

Listing 3 shows the assembly code of *CMjpegDec::-Transform* method that receives a media sample (i.e., *IMediaSample *pIn*) and outputs another media sample (i.e., *IMediaSample *pOut*), with the image decompression carried out at line 7 by the function *DecompressBegin* that receives as arguments an output YCbCr buffer, an input JPEG image buffer and an image header info.

3.3 Listing 3 *CMjpegDec::Transform* method partial assembly code

1 **mov [ebp**+ms_exc.disabled], 1 **push dword ptr [esi**+0C0h] **call**
2 _DecompressEnd@4 *; DecompressEnd(x) push [ebp+pInfoHeader] ;*
3 image header **push edi** *; image src buffer push*
4 **dword ptr [esi**+0C0h] *; image dst buffer call _DecompressBegin@12*
5 *; DecompressBegin(x,x,x) ; decompress frame or*
6 **[ebp**+ms_exc.disabled], 0FFFFFFFFh **and [ebp**+var_6C], 0

Listing 3 shows the assembly code of CMjpegDec::-Transform method that receives a media sample (i.e., IMediaSample ∗pIn) and outputs another media sample (i.e., IMediaSample ∗pOut), with the image decompression carried out at line 7 by the function DecompressBegin that receives as arguments an output YCbCr buffer, an input JPEG image buffer and an image header info.

Listing 4 partially shows the disassembling of *DecompressBegin* function indicating that *MJpegDec* filter uses IJG JPEG library during decompression process. Deeper analysis of this function revealed through the C routines into the library that it's a generic JPEG decompressor implementation full of variations and consequently with a severe dead code overhead.

3.4 Listing 4 Partial disassembling of *DecompressBegin* function

1 **lea edi, [esi**+1F8h] **lea eax, [esi**+8] **push eax call**
2 _jpeg_exception_error@4 *; jpeg_exception_error(x) mov [edi], eax*
3 **push edi call** _jpeg_create_decompress@4 ;
4 jpeg_create_decompress(x) **mov byte ptr [esi**+399h], 1 **mov**
5 **eax, [ebx**+10h] **test eax, eax jnz** loc_74823996

Complementing this information with a reverse engineering on the SDK source code, Figures 2, 3 and 4 describe the DirectShow static architecture.

4 Video surveillance domain engineering

We'll follow the usual approach for SPL development organized in domain engineering and application engineering, with the former also defined as "development for reuse" and split into domain analysis, domain design and domain implementation. Application engineering is also defined as "development with reuse". SPL technology consists of a set of tools, methods and techniques to create a set of products, software systems, from a common base for all products (i.e., all system settings) [13]. Furthermore, to bridge the gap between variability model and model elements and automate the mapping of system configuration to the domain model and generating the final code, we followed the MDD process proposed in [16] as showing in Figure 5.

4.1 The domain analysis

The domain analysis is split in domain scope that determines which systems and features are part of video surveillance domain (Figure 6) and feature modeling which identifies the commonality and variability as well as the relationship among features in the variability model. Feature modeling was proposed as a part of the Feature Oriented Domain Analysis (FODA) [21], and since then, it is applied in various fields. The model features are visually represented by functionality diagrams that model system variability notwithstanding the used implementation mechanism, such as inheritance, templates or conditional compilation.

For the video surveillance domain, several feature diagrams are provided—one for the filter graph or pipeline and one for each type of filter (Figures 7, 8, 9 and 10).

The feature diagram of the filter graph, Figure 7, identifies and classifies all filters that compose the filter graph as well as all possible links between them. The root node represents the concept (i.e., filter graph) that consists of three features, one for each type of filters that make up the filter graph (i.e., source, transform and renderer filters). The source filter concept contains two alternative features, live device or file, meaning that in any video surveillance system instance, only one of them should be included. The transform filter concept consists of two or-features (i.e., codecs and Processing), meaning that any video surveillance system instance has at least one of them.

As indicated by the cardinalities [0..*] and [1..*], the transform filters are optional and a filter graph can present none, one or several of them, and source and rendering filters are mandatory, i.e., a filter graph or pipeline must present at least one of each.

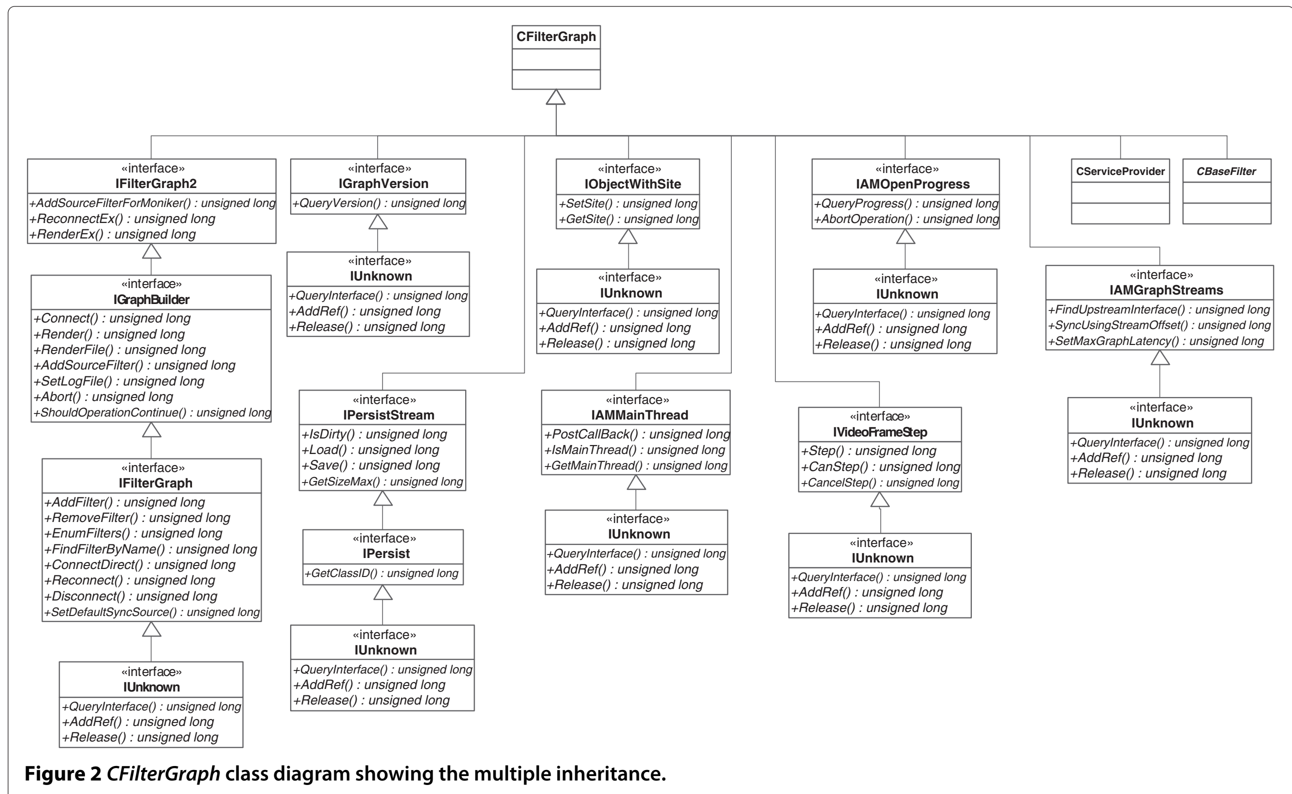

Figure 2 ***CFilterGraph*** **class diagram showing the multiple inheritance.**

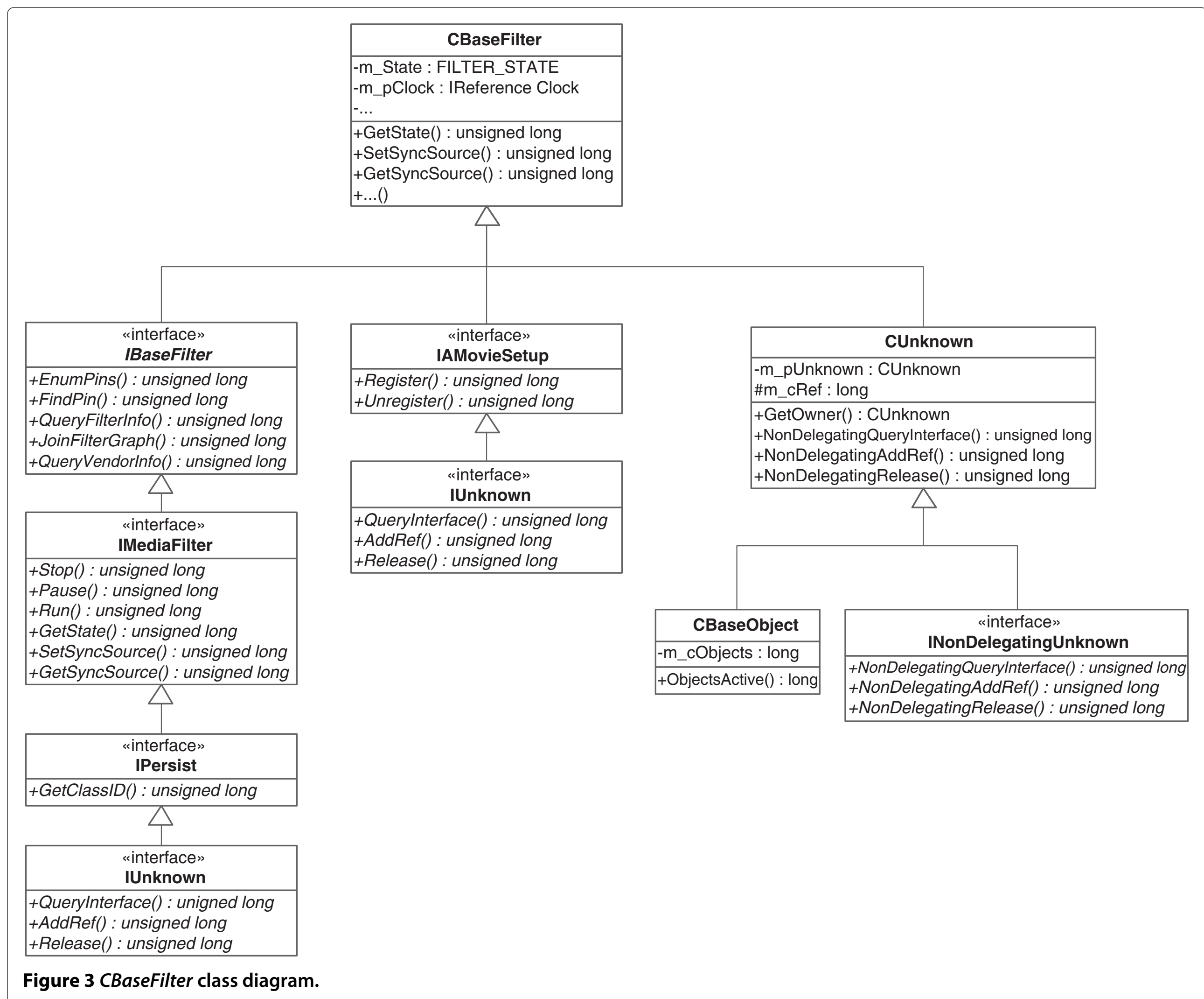

Figure 3 *CBaseFilter* **class diagram.**

All the above feature diagrams are internally represented using XML notation as can be shown in Listings 5 and 6 for the description of the File renderer filter and filter graph, respectively.

4.2 Listing 5 XML Feature diagram of a renderer filter designed File

```
1  <?xml version="1.0" ?> <feature name="File">
2      <feature name="Localization" type="Mandatory">
3          <featureGroup type="Alternative">
4              <feature name="Local" type="Mandatory"/>
5              <feature name="Remote" type="Mandatory">
6                  <featureGroup type="Alternative">
7                      <feature name="SMB" type="Mandatory"/>
8                      <feature name="FTP" type="Mandatory"/>
9                  </featureGroup>
10             </feature>
11         </featureGroup>
12     </feature>
13     <feature name="Format" type="Mandatory">
14         <featureGroup type="Alternative">
15             <feature name="AVI" type="Mandatory"/>
16             <feature name="WAV" type="Mandatory"/>
17             <feature name="OWN" type="Mandatory"/>
18         </featureGroup>
19     </feature>
20 </feature>
```

4.3 Listing 6 Filter graph XML Feature diagram

```
1  <?xml version="1.0" ?> <feature name="Filter Graph">
2    <feature min="1"max="*"name="Source Filter"type="Mandatory">
3        <featureGroup type="Alternative">
4          <feature name="Live Device" type="Mandatory">
5              <featureGroup type="Alternative">
6                <feature name="Analogic Camera" type="Mandatory"/>
7                <feature name="Microphone" type="Mandatory"/>
8                <feature name="IP Camera" type="Mandatory"/>
9              </featureGroup>
10         </feature>
11           ...
12       </featureGroup>
13   </feature>
14   ...
15   <feature min="1"max="*"name="Renderer Filter"
type="Mandatory">
16       <featureGroup type="Or">
17         <feature name="Transmission" type="Mandatory">
18             <featureGroup type="Alternative">
19               <feature name="HTTP" type="Mandatory"/>
20               <feature name="RTSP" type="Mandatory"/>
21             </featureGroup>
22         </feature>
23           ...
24       </featureGroup>
```

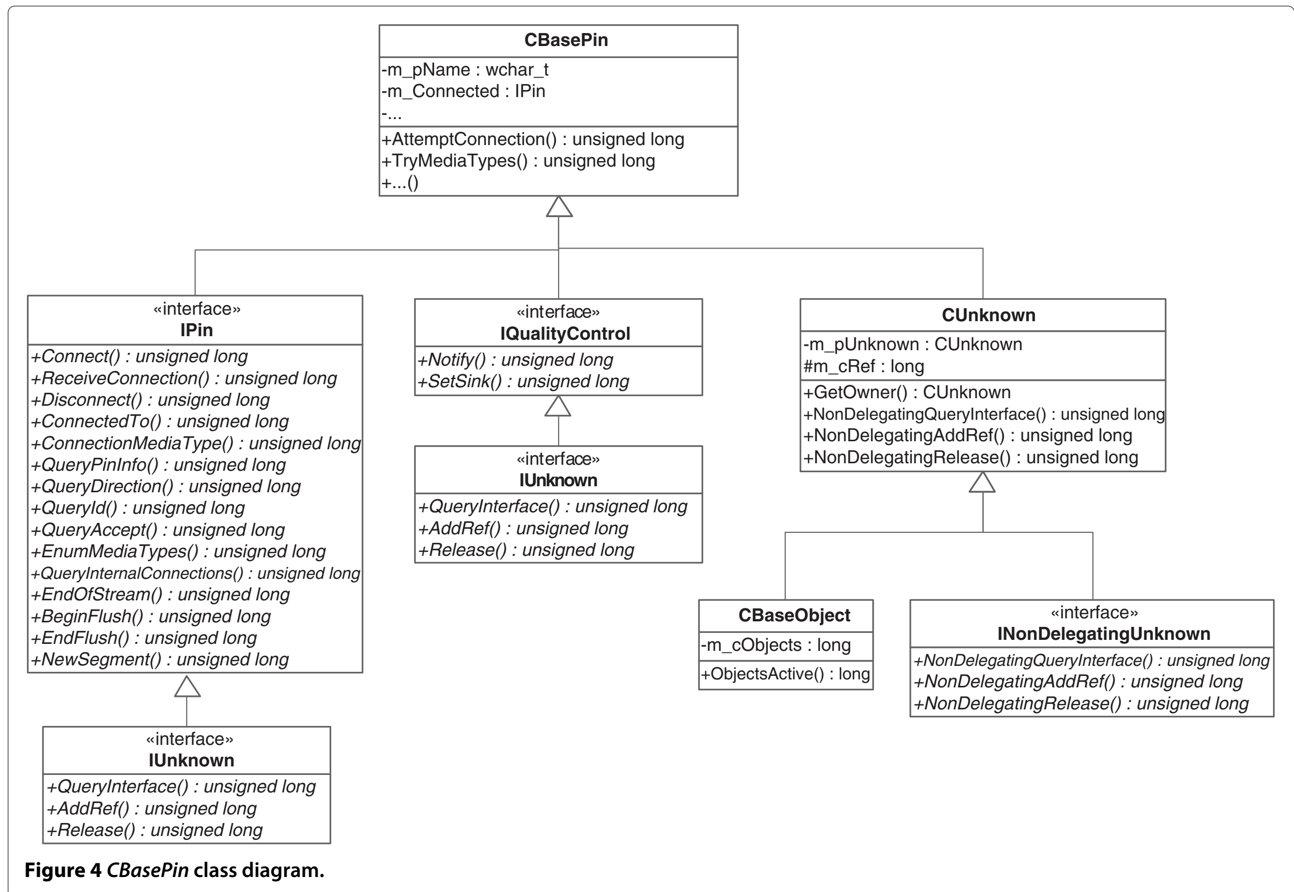

Figure 4 ***CBasePin*** **class diagram.**

```
25    </feature>
26  </feature>
```

4.4 The domain design and implementation

Domain design is the next step after the specification of applications and components in video surveillance domain and its main goal is developing a common architecture for the video surveillance family of products. Having defined the common architecture, the implementation domain proposes developing configurable and customizable components to support the implementation of the video surveillance system domain architecture [22]. The implemented generative-friendly framework for video surveillance software product family, basically, handles variability at two levels of granularity (intra-filter and inter-filter) and consists of:

(i) Several and different artifact templates encoded in C++ template metaprogramming for each type of filter that is generated;
(ii) A filter graph artifact template describing the pipeline as a confederation of filters based on component technology and encoded in C++ template metaprogramming;
(iii) A set of meta-expressions and presence conditions with direct correspondence to the artifact templates and their variability points, and expressed in terms of features used to map features in feature models to the artifact templates, and so, giving semantics to those features [23];
(iv) A set of implicit rules for filters connection and validation, are internal and implicitly defined as part of filter graph within its built-in *Intelligent Connector*. These rules are also encoded in C++ metaprogramming artifact templates.

4.4.1 The filter artifact template

The filter artifact template does not implement an artifact template for each filter type, in opposition to other filter implementations that classify each filter by category using specific classes for the purpose. Instead, all the filters follow the same implementation that allows the representation of artifact filters in a more flexible manner. This methodology was used to allow easier refactoring of legacy code from external libraries that implements some filters functionalities. The framework allows exploring the granularity at two levels: coarse-grained and fine-grained. A coarse-grained granularity allows changing between filters; as an example, changing a JPEG transformation filter by a H.264 transformation filter. A fine-grained granularity allows having different instances of the same

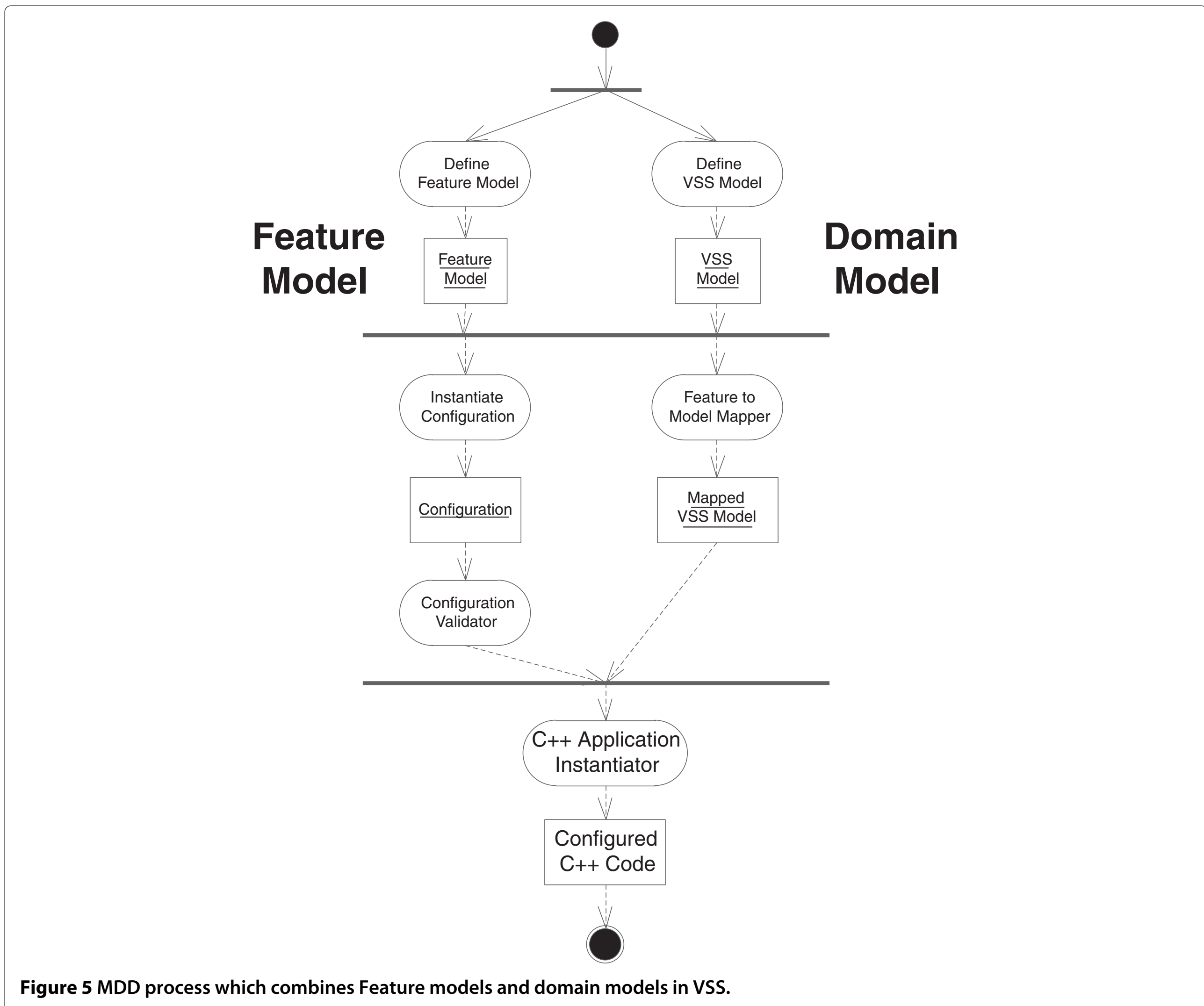

Figure 5 MDD process which combines Feature models and domain models in VSS.

filter to satisfy the system requirements, e.g., the JPEG transformation filter can have a generic implementation and an optimized implementation.

To develop a new filter artifact template, the unique requisites that the filter needs to implement are:

(i) Define the data type named *type* that informs what is the type of the filter (source, transform, renderer);
(ii) The *Initialize* and *Cleanup* methods that initialize and cleanup all the filter resources;
(iii) The *Connect* method that allows connecting other filter to this filter;
(iv) The *Start*, *Stop* and *Pause* methods that control the execution data flow;
(iv) The *Process* method that process data.

Most of the artifact templates related to different kind of filters are automatically generated running an online partial evaluator on legacy codes from previous video surveillance projects (external libraries). Such refactoring process was implemented based on an automatic partial evaluation for the C++ language named transformer [13] on a first stage and then the output was adapted to our filter template architecture.

In the following sub-section, two filter artifact implementations are presented, one for an input filter and the other for a transformation filter.

4.4.2 The input filter artifact template

Now let's take the input filter *V4L2* as represented by the feature diagram in Figure 8 to partially describe the artifact template of such kind of filters. Due to huge extension of the code we'll focus on the filter Cleanup method needed to free all allocated resource by the filter. Three created structures, one for each capture method and a skeleton artifact template for *CV4L2Source* along with the *Cleanup* method are created as shown by Listings 7 and 8, respectively.

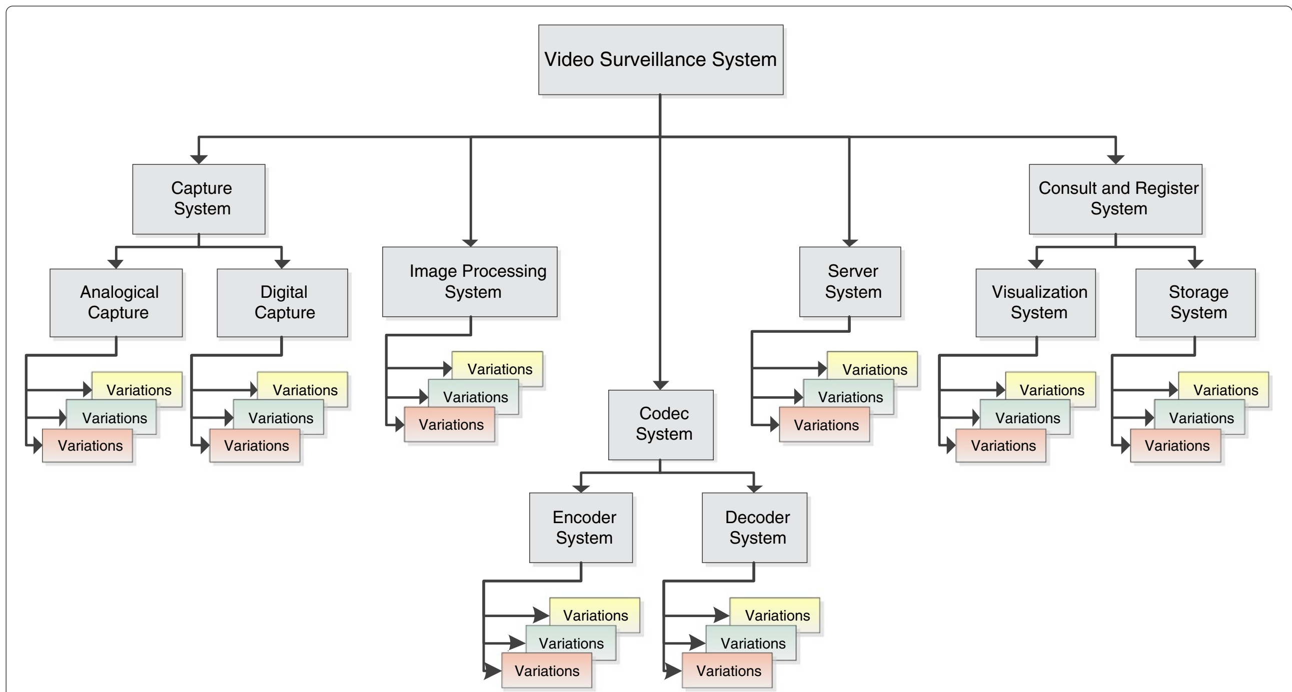

Figure 6 Video Surveillance Domain Scoping.

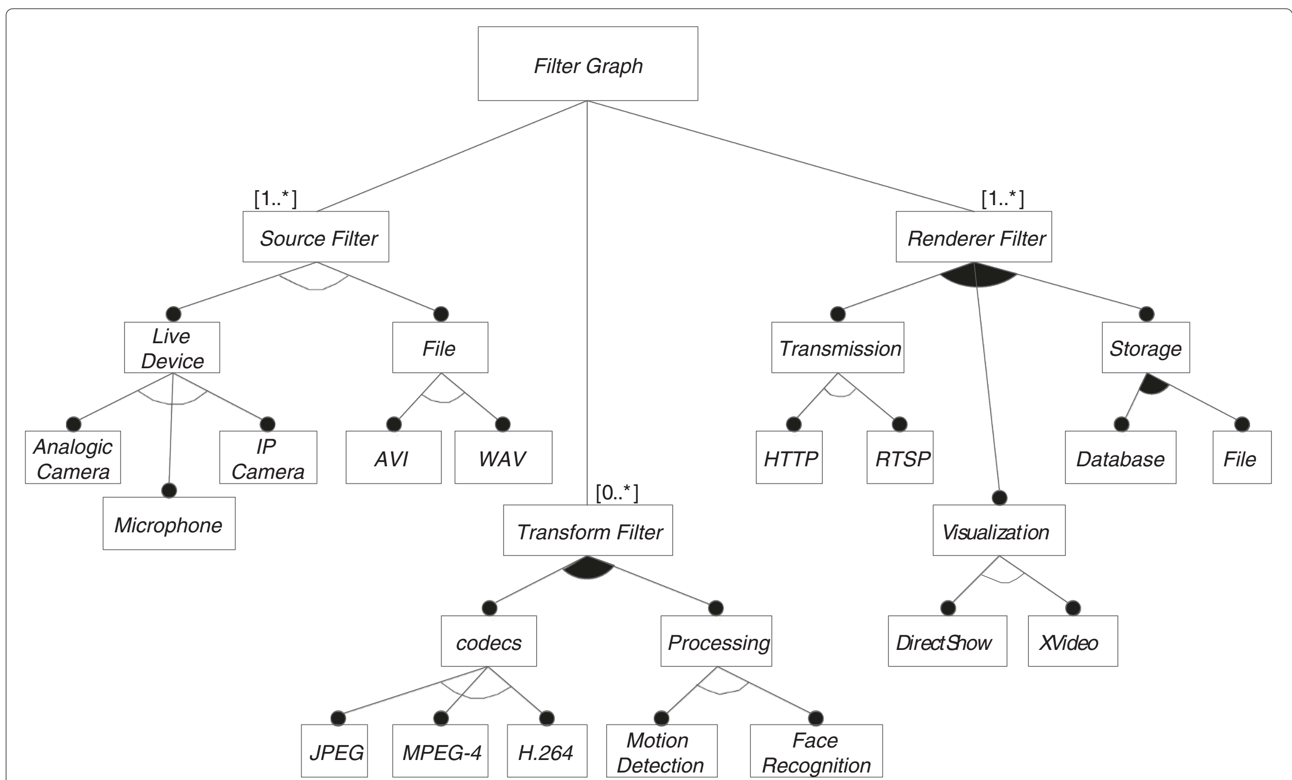

Figure 7 Filter graph feature diagram.

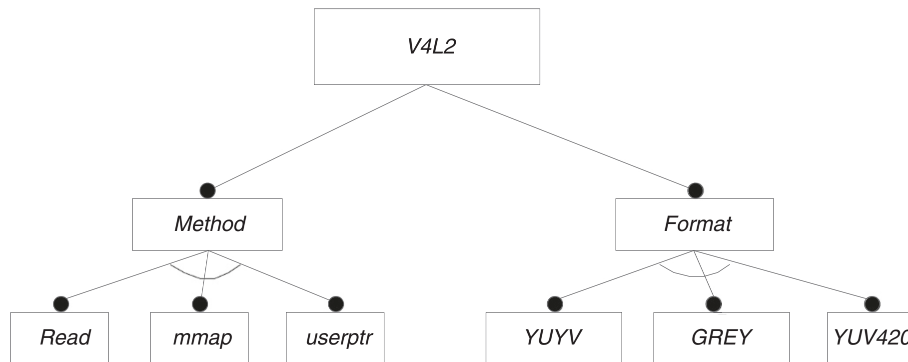

Figure 8 **Feature diagram of a V4L2 source filter.**

4.5 Listing 7 Data type definition for the capture mode

```
1  // read method
2  struct v4l2_read_method_t {};
3
4  // mmap method
5  struct v4l2_mmap_method_t {};
6
7  // userptr
8  struct v4l2_userptr_method_t {};
```

4.6 Listing 8 *CV4L2* class skeleton artifact template code

```
1  template <typename IOMethod>
2  class CV4L2Source {
3  public:
4      // defines the filter type
5      typedef source_filter_t type;
6
7      CV4L2Source() {}
8      CV4L2Source() {}
9
10     // TODO: add the other class methods
11     void Cleanup();
12
13 public:
14     // TODO: add class variables
15 };
16
17 // cleanup method
18 void CV4L2Source::Cleanup() {
19     CV4L2SourceImpl<IOMethod>::Cleanup<
   CV4L2Source>(*this);
20 }
```

The *Cleanup* method uses a template data structure, *CV4L2SourceImpl* (see Listing 9), which defines three specialized templates, one for each capture mode. Based on the selected capture mode, one of these templates

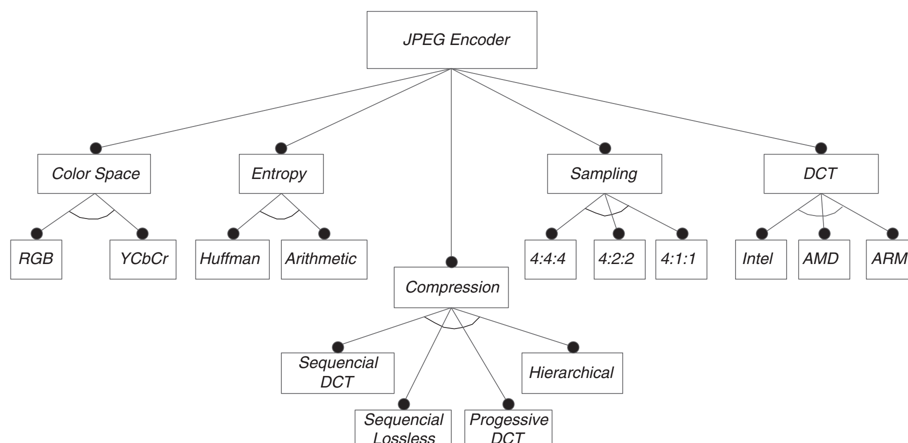

Figure 9 **Feature diagram of a transform filter for JPEG decompression.**

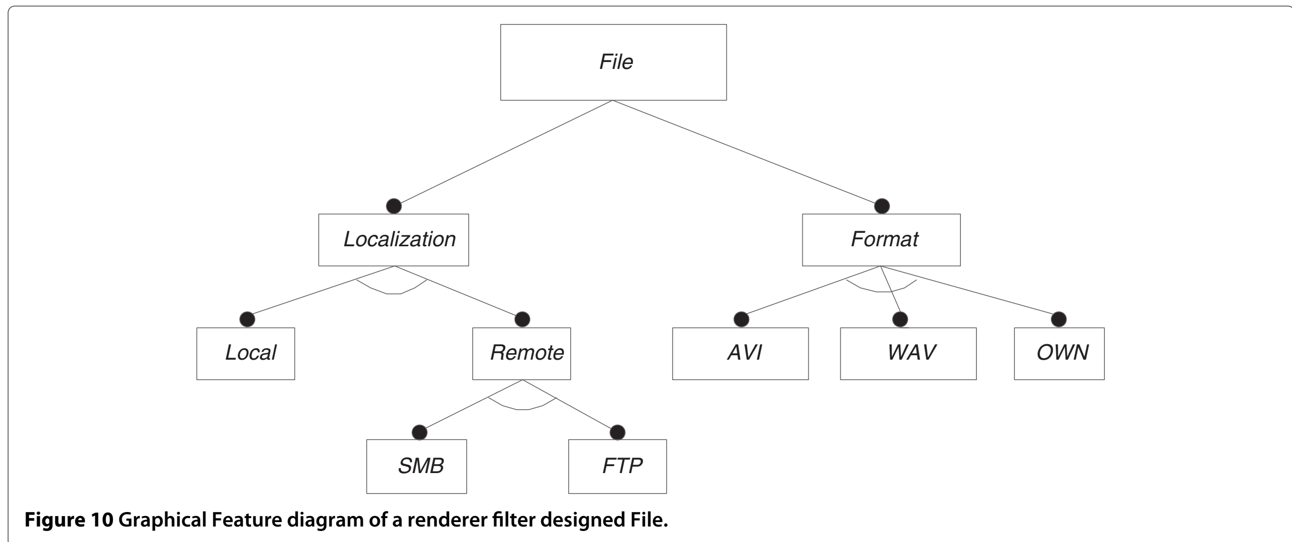

Figure 10 Graphical Feature diagram of a renderer filter designed File.

is instantiated with an inline *Cleanup* function whose body is statically patched into the place where *Cleanup* is supposed to be called.

4.7 Listing 9 *CV4L2SourceImpl* template code structure

```
1  template <typename IOMethod> struct
   CV4L2SourceImpl;
2
3  template<>
4  struct CV4L2SourceImpl<v4l2_read_method_t> {
5      template<typename T>
6      static inline void Cleanup(T &t) {
7          // TODO: insert the code to cleanup using read
   method
8      }
9  };
10
11 template<>
12 struct CV4L2SourceImpl<v4l2_mmap_method_t> {
13     template<typename T>
14     static inline void Cleanup(T &t) {
15         // TODO: insert the code to cleanup using
   mmap method
16     }
17 };
18
19
29 template<>
21 struct CV4L2SourceImpl<v4l2_userptr_method_t>
   {
22     template<typename T>
23     static inline void Cleanup(T &t) {
24         // TODO: insert the code to cleanup using
   userptr method
25     }
26 };
```

Finally, a concrete instance of the *CV4L2Source* artifact template using one of the capture modes (i.e., *read_method*) is given in Listing 10.

4.8 Listing 10 Instantiation of *CV4L2Source* class with *read* capture mode

```
1 CV4L2Source<v4l2_read_method_t> v4l2;
2 V4l2.Cleanup();
```

4.8.1 The transform filter artifact template

Taking the diagram in Figure 9 let's partially describe the artifact template related to the concept *JPEG Decoder* that decompresses images streaming in JPEG format to raw format (e.g., YCbCr or RGB). Again due to the code huge extension we'll only discuss the implementation of sampling and color Space features. Three data type will be defined, one for each JPEG's sampling, 4:4:4. 4:2:2 and 4:1:1 (Listing 11) as well as their associated specialized template structure *CJPEGSamplingImpl* as shown in Listing 12.

4.9 Listing 11 JPEG's sampling data type definition

```
1 // 4:4:4 sampling
2 struct jpeg_444_sampling_t {};
3
4 // 4:2:2 sampling
5 struct jpeg_422_sampling_t {};
6
7 // 4:1:1 sampling
8 struct jpeg_411_sampling_t {};
```

4.10 Listing 12 Template struct code for *CJPEGSamplingImpl*

```
1  template <typename SamplingType> struct
   CJPEGSamplingImpl;
2
3  template<>
4  struct CJPEGSamplingImpl<jpeg_444_sampling_t>
   {
5     template<typename T>
6     static inline void apply(T &t) {
7       // TODO: insert the code for sampling 4:4:4
8     }
9  };
10
11 template<>
12 struct CJPEGSamplingImpl<jpeg_422_sampling_t>
   {
13    template<typename T>
14    static inline void apply(T &t) {
15      // TODO: insert the code for sampling 4:2:2
16    }
17 };
18
19 template<>
20 struct CJPEGSamplingImpl<jpeg_411_sampling_t>
   {
21    template<typename T>
22    static inline void apply(T &t) {
23      // TODO: insert the code for sampling 1:1:1
24    }
25 };
```

Each specialized template defines an inline function *apply* that implements the related sampling functionality. Upon instantiation, only one of the specialized templates is instantiated by the compiler, depending on the type of data *SamplingType*. The *Color Space* concept implementation follows similar steps (Listings 13 and 14) but as it includes alternative features, YCbCr and RGB, two different types of data should be created as illustrated in Listing 13.

4.11 Listing 13 Definition of data types for JPEG's Color implementation

```
1  // YCbCr color space
2  struct jpeg_ycbcr_colorspace_t {};
3
4  // RGB color space
5  struct jpeg_rgb_colorspace_t {};
```

4.12 Listing 14 *CJPEGColorSpaceImpl* template structure code

```
1  template <typename ColorSpaceType> struct
   CJPEGColorSpaceImpl;
2
3  template<>
4  struct
   CJPEGColorSpaceImpl<jpeg_ycbcr_colorspace_t> {
5     template<typename T>
6     static inline void apply(T &t) {
7       // TODO: insert the code for color space YCbCr
8     }
9  };
10
11 template<>
12 struct
   CJPEGColorSpaceImpl<jpeg_rgb_colorspace_t> {
13    template<typename T>
14    static inline void apply(T &t) {
15      // TODO: insert the code for color space RGB
16    }
17 };
```

Now the *CJPEGDec* artifact template can be presented (see Listing 15) and should be later instantiated with two template input parameters given by the data types of *SamplingType* and *Color Space* features. *CJPEGDec::Decode( )* takes as input parameter a buffer containing the image information stored in JPEG format. In Listing 16 sampling 4:4:4 and RGB color space are chosen.

4.13 Listing 15 *CJPEGDec* is the template artifact related to *JPEG Decoder* concept

```
1  template <typename SamplingType, typename
   ColorSpaceType>
2  class CJPEGDec
3  {
4  public:
5     // defines the filter type
6     typedef transform_filter_t type;
7
8     CJPEGDec() {}
9     ~CJPEGDec() {}
10
11    // TODO: add the other class methods
12
13    void Decode(CBuffer *pBuffer);
14
15 public:
16    // TODO: add class variables
17 };
18
19 // decode method
20 template <typename SamplingType, typename
   ColorSpaceType>
21 void CJPEGDec::Decode(CBuffer *pBuffer)
22 {
```

```
23     //...
24
25     CJPEGSamplingImpl<ColorSpaceType>::apply<
   CJPEGDec>(*this);
26
27     CJPEGColorSpaceImpl<SamplingType>::apply<
   CJPEGDec>(*this);
28 }
```

4.14 Listing 16 Instantiation example of the artifact template *CJPEGDec*

```
1 // image buffer
2 CBuffer buffer;
3
4 CJPEGDec<jpeg_444_sampling_t,
  jpeg_rgb_colorspace_t> dec;
5 dec.Decode(&buffer);
```

4.14.1 The filter graph artifact template

To implement the *CFilterGraph* class, specialized templates are used to instantiate an appropriate artifact template which creates an instance of filter graph with a predefined number of filters. The *CFilterGraph* artifact template receives a Boost MPL [24,25] vector with the data types of all filters and the number of filters as its input parameters. The number and data type of each filter depends on the configuration required to implement the desired functionality and the simplest filter graph configuration contains at least two types of filters: a source filter and a renderer filter (see Listing 17).

4.15 Listing 17 Creating a *CFilterGraph* instance using a data type, *Types*, representing a vector with 2 filters: *CV4L2Source* and *CXVRenderer*

```
1 typedef mpl::vector<CV4L2Source, CXVRenderer>
  Types;
2 CFilterGraph<Types, mpl::size<Types>::value> fg;
```

By default nineteen specialized templates are created, allowing at least the instantiation of a *CFilterGraph* with at least two and at most twenty filters with the number of specialized templates reconfigured by changing the compilation variable MAX_FILTER-GRAPH_SPECIALIZATIONS (see Listing 18).

4.16 Listing 18 A snippet of *CFilterGraph* C++ template metaprogramming artifact template

```
1 template <typename FilterTypes, int N> class
  CFilterGraph;
2
3 template <typename FilterTypes>
4 class CFilterGraph<FilterTypes, 2> {
5     typedef typename mpl::at<FilterTypes,
  mpl::long_<0> >::type Filter0;
6     typedef typename mpl::at<FilterTypes,
  mpl::long_<1> >::type Filter1;
7
8     Filter0_f0;
9     Filter1_f1;
10 public:
11     Filter0& const GetFilter(mpl::int_<0>)
  {return_f0;}
12     Filter1& const GetFilter(mpl::int_<1>)
  {return_f1;}
13 };
14
15 template <typename FilterTypes>
16 class CFilterGraph<FilterTypes, 3> {
17     typedef typename mpl::at<FilterTypes,
  mpl::long_<0> >::type Filter0;
18     typedef typename mpl::at<FilterTypes,
  mpl::long_<1> >::type Filter1;
19     typedef typename mpl::at<FilterTypes,
  mpl::long_<2> >::type Filter2;
20
21     Filter0_f0;
22     Filter1_f1;
23     Filter2_f2;
24 public:
25     Filter0& const GetFilter(mpl::int_<0>)
  {return_f0;}
26     Filter1& const GetFilter(mpl::int_<1>)
  {return_f1;}
27     Filter2& const GetFilter(mpl::int_<2>)
  {return_f2;}
28 };
29 ...
```

To access each type of filters that make up the filter graph, the *CFilterGraph* class implements the method *GetFilter* which receives as input parameter the Boost MPL data type *int_* with the index of the filter and returns the data type of the filter. To simplify and also offer some flexibility to the implementation of each *CFilterGraph* template specializations, a few macros based on the token past operator are used, such as DEFINE_FILTER_TYPE(n), and CREATE_FILTER_GRAPH(n) to define the data types of each filter at index n in the vector of types and create each specialization with n filters, respectively (Listings 19 and 20.

4.17 Listing 19 Macro that defines the type of filters: *n* is the index in the vector of types

```
1 #define DEFINE_FILTER_TYPE(n)\
2     typedef typename mpl::at<TList,
  mpl::long_<n> >::type Filter##n;
```

```
3 #define DEFINE_FILTER_TYPE_1(TList)
  DEFINE_FILTER_TYPE(0)
4 #define DEFINE_FILTER_TYPE_2(TList)
  DEFINE_FILTER_TYPE_1(TList)\
5     DEFINE_FILTER_TYPE(1)
6 #define DEFINE_FILTER_TYPE_3(TList)
  DEFINE_FILTER_TYPE_2(TList)\
7     DEFINE_FILTER_TYPE(2)
8 ...
```

4.18 Listing 20 Macro that creates specialized templates of *CFilterGraph*: *n* is the number of filters

```
1 #define CREATE_FILTER_GRAPH(n)\
2     template <typename FilterTypes>\
3     class CFilterGraph<FilterTypes, n>\
4     {\
5         DEFINE_FILTER_TYPE_##n(FilterTypes);\
6         FILTER_TYPE_##n;\
7     public:\
8      GET_FILTER_##n;\
9     };
```

To create the class variables of each filters and the filter graph *GetFilter* methods, similar macros are also used and the final implementation of *CFilterGraph* artifact template will be as shown in Listing 21.

4.19 Listing 21 Final implementation of the *CFilterGraph* artifact template

```
1 template <typename FilterTypes, int N> class
  CFilterGraph;
2
3 CREATE_FILTER_GRAPH(2);
4 CREATE_FILTER_GRAPH(3);
5 ...
6 CREATE_FILTER_GRAPH(20);
```

Listing 22 shows an example of video image capture from an ONVIF compliant IP camera. To capture the images from the IP camera, the input filter RTSP (*CRTSP-Source*) is used, as the IP camera sends images using the JPEG compressed video format, the JPEG transform filter (*CJPEGDec*) is used and the xvideo renderer filter allows seeing the images on the screen (*CXVRenderer*. To control and receive camera events a filter that implements web services is used. For that purpose several auxiliary artifact templates were implemented to validate and connect all the filters following a pipeline of streams that compose a filter graph artifact template. The connection between filters is represented by a vector of connections, *Connections*, in which each link is an ordered pair consisting of the index into the vector of filter types and the respective filter type.

4.20 Listing 22 Ordered pair vector specifying connections between *CRTSPSource*, *CJPEGDec* and *CXVRenderer* filters

```
1  // filter types vector
2  typedef mpl::vector<CRTSPSource, CJPEGDec,
   CXVRenderer> Types;
3
4  ...
5  // pair that represents the link between
   CRTSPSource and CJPEGDec filter
6  typedef mpl::pair<mpl::int_<0>,mpl::int_<1>> c1;
7  // pair that represents the link between CJPEGDec
   and CXVRenderer filter
8  typedef mpl::pair<mpl::int_<1>,mpl::int_<2>> c2;
9
10 // filter connections vector
11 typedef mpl::vector<c1, c2> Connections;
```

Listings 23, 24 and 25 specify how the links described by the *Connections* vector are applied during the filter graph construction by an auxiliary function template, *ConnectFilters* (see Listing 24), which receives a *Connection* vector as input parameter. At the instantiation time, a specialized or concrete *CFilterGraph* class should implements the *ConnectFilters* function template which uses the inline *ConnectFilters* of the *FilterGraphManager* structure given by Listing 25.

4.21 Listing 23 Building a filter graph stream by connecting a set of filters

```
1 // filter graph variable
2 CFilterGraph<Types, mpl::size<Types>::value> fg;
3 // connect filters
4 fg.ConnectFilters<Connections>();
```

4.22 Listing 24 *CFilterGraph::ConnectFilters(...)* implementation

```
1  template <typename FilterTypes>
2  class CFilterGraph<FilterTypes, 2>
3  {
4     ...
5  public:
6      template <typename FilterConnections>
7      void ConnectFilters() {
8        FilterGraphManager<mpl::size<FilterConnec-
   tions>::value>::template
9            ConnectFilters<FilterTypes,
   FilterConnections,CFilterGraph>(*this);
10      }
11 };
```

4.23 Listing 25 Template struct *FilterGraphManager* implemented as a filter graph manager

```
1 template <int N> struct FilterGraphManager;
```

```
2
3 template <int N>
4 struct FilterGraphManager {
5    template <typename Types, typename Connections, typename FilterGraph>
6    inline static void ConnectFilters(FilterGraph& fg) {
7     // TODO: add code to connect filters recursivelly
8    }
9
10   template <typename FilterGraph>
11   inline static void StartFilters(FilterGraph& fg) {
12    // TODO: add code to start filters recursivelly
13   }
14
15   template <typename FilterGraph>
16   inline static void StopFilters(FilterGraph& fg) {
17    // TODO: add code to stop filters recursivelly
18   }
19
20   template <typename FilterGraph>
21   inline static void PauseFilters(FilterGraph& fg) {
22    // TODO: add code to pause filters recursivelly
23   }
24 };
25
26 template<>
27 struct FilterGraphManager <0> {
28    template <typename Types, typename Connections, typename FilterGraph>
29    inline static void ConnectFilters(FilterGraph& fg) { }
30
31    template <typename FilterGraph>
32    inline static void StartFilters(FilterGraph& fg){}
33
34    template <typename FilterGraph>
35    inline static void StopFilters(FilterGraph& fg) { }
36
37    template <typename FilterGraph>
38    inline static void PauseFilters(FilterGraph& fg){}
39 };
```

The *FilterGraphManager* template structure in Listing 25 implements the management of certain filter graph operations, such as, filter connections and *start*, *stop*, and *pause* of the filter streams execution. The inline functions *ConnectFilter*, *StartFilters*, *StopFilters* and *PauseFilters* connects two filters of a given filter graph, starts the filters execution flow, stops the filters execution flow and pauses the filters execution flow, respectively.

Since the management operations are applied recursively with the number of operations given by *N*, the *FilterGraphManager* structure is implemented by a generic template (line 05 to 28) and a specialized one with input parameter zero (line 31 to 45) to indicate the termination condition. When *N* is zero, all inline functions defined inside the specialized template will present an empty body and the C++ compiler patches no code at all.

The *ConnectFilter* inline function of *FilterGraphManager* structure accepts three template parameters (i.e., *Types*, *Connections*, *FilterGraph*) and a filter graph reference as input parameter to connect the filters composing a filter graph as shown by its implementation in Listing 26.

4.24 Listing 26 Inline *ConnectFilters* function code

```
1 template <int N> struct FilterGraphManager;
2
3 template <int N>
4 struct FilterGraphManager {
5    template <typename Types, typename Connections, typename FilterGraph>
6    inline static void ConnectFilters(FilterGraph& fg) {
7       // call ConnectFilters for N-1 index
8       FilterGraphManager<N-1>::ConnectFilters <Types,Connections,FilterGraph>(fg);
9
10      // aux data types
11      typedef typename mpl::at<Connections,mpl::long_<N-1> >::type elem;
12      typedef typename mpl::at<Types, mpl::first<elem>::type >::type
13                  out_filter_type;
14      typedef typename mpl::at<Types, mpl::second<elem>::type >::type
15                  in_filter_type;
16
17     CheckErrorOnConnection<AllowConnection< out_filter_type,
18                  in_filter_type>::value, out_filter_type, in_filter_type>;
19
20      // connect filter first<elem> to second<elem> ; elem<A,B>
21      fg.GetFilter(mpl::first<elem>::type()).Connect(&fg.GetFilter(mpl::second<elem>::type()),
22                  fg.GetFilter(mpl::second<elem> ::type()).Process);
23    }
24
25    // more inline functions
26 };
```

```
27
28 template<>
29 struct FilterGraphManager <0> {
30     template <typename Types, typename
   Connections, typename FilterGraph>
31     inline static void ConnectFilters(FilterGraph&
   fg) { }
32
33     // more inline empty functions
34 };
```

In each iteration is called first the same function recursively with value *N-1* (line 11), then a test is run to validate the connection between two filters (lines 20 to 21) and finally the connection is made (line 24 to 25). As an example, consider the filter graph, the code to create it and the recursive iteration inside *ConnectFilters* method for the filter graph *fg* as shown in Figures 11, 12 and Listing 27, respectively.

4.25 Listing 27 Filter graph code builder for Figure 11

```
1 // filter types vector
2 typedef mpl::vector<f1, f2, f3, f4, f5> Types;
3
4 // links between filters
5 typedef mpl::pair<mpl::int_< 0 >,mpl::int_< 1 > >
  c12;
6 typedef mpl::pair<mpl::int_< 0 >,mpl::int_< 2 > >
  c13;
7 typedef mpl::pair<mpl::int_< 1 >,mpl::int_< 3 > >
  c24;
8 typedef mpl::pair<mpl::int_< 2 >,mpl::int_< 4 > >
  c35;
9
10 // filter connections vector
11 typedef mpl::vector< c12, c13, c24, c35 >
   Connections;
12
13 // filter graph variable
14 CFilterGraph<Types, mpl::size<Types>::value> fg;
15 // connect filters
16 fg.ConnectFilters<Connections>();
```

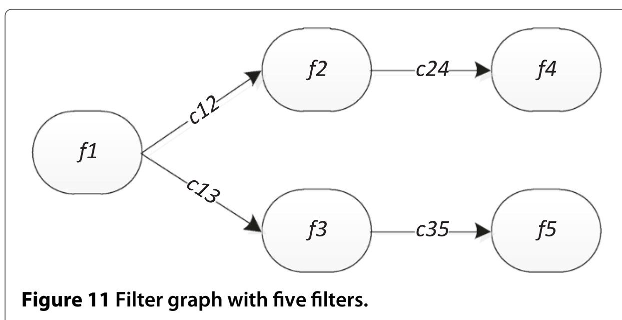

Figure 11 Filter graph with five filters.

The *StartFilters*, *StopFilters* or *PauseFilters* inline functions template of the *FilterGraphManager* structure given by Listings 28, Listing 29 Inline *StopFilters* function code and 30, performs only two operations. The first one will recursively call the *Start*, *Stop* or *Pause* methods of filters at indexes lower than *N-1* and greater than zero while the second one executes the *Start*, *Stop* or *Pause* methods of the filter at index *N-1*.

4.26 Listing 28 Inline *StartFilters* function code

```
1 template <int N> struct FilterGraphManager;
2
3 template <int N>
4 struct FilterGraphManager {
5     template <typename FilterGraph>
6     inline static void StartFilters(FilterGraph& fg) {
7         // call StartFilters for N-1 index
8         FilterGraphManager<N-
  1>::StartFilters<FilterGraph>(fg);
9
10        // start filter[N-1]
11        fg.GetFilter(mpl::int_<N-1>()).Start();
12    }
13 };
14
15 template<>
16 struct FilterGraphManager< 0 > {
17     template <typename FilterGraph>
18     inline static void StartFilters(FilterGraph& p) { }
19 };
```

4.27 Listing 29 Inline *StopFilters* function code

```
1 template <int N>
2 struct FilterGraphManager {
3     template <typename FilterGraph>
4     inline static void StopFilters(FilterGraph& fg) {
5         // call stop_filters for N-1 index
6         FilterGraphManager<N-
  1>::StopFilters<FilterGraph>(fg);
7
8         // stop filter[N-1]
9         fg.GetFilter(mpl::int_<N-1>()).Stop();
10        }
11 };
12
13 template<>
14 struct FilterGraphManager<0> {
15     template <typename FilterGraph>
16     inline static void StopFilters(FilterGraph& p) { }
17 };
```

4.28 Listing 30 Inline *PauseFilters* function code

```
1 template <int N> struct FilterGraphManager;
2
3 template <int N>
```

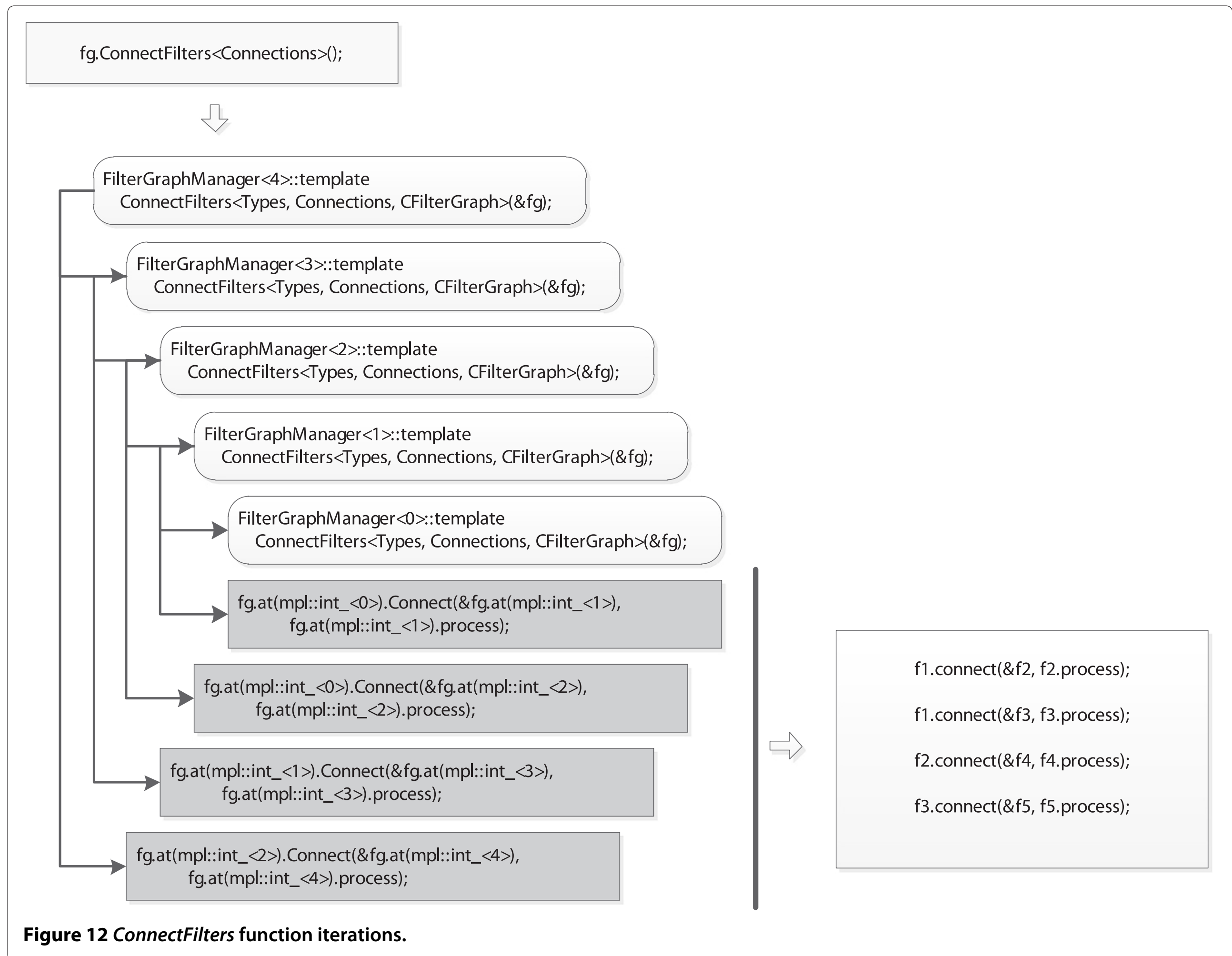

Figure 12 ***ConnectFilters*** **function iterations.**

```
4  struct FilterGraphManager {
5      template <typename FilterGraph>
6      inline static void PauseFilters(FilterGraph& fg) {
7          // call pause_filters for N-1 index
8          FilterGraphManager<N-
   1>::PauseFilters<FilterGraph>(fg);
9
10         // pause filter[N-1]
11         fg.GetFilter(mpl::int_<N-1>()).Pause();
12     }
13 };
14
15 template<>
16 struct FilterGraphManager<0> {
17     template <typename FilterGraph>
18     inline static void PauseFilters(FilterGraph& p) { }
19 };
```

4.28.1 The intelligent connect artifact template

The *AllowConnection* artifact template, shown in Listing 31, was implemented to validate the connection between two filters based on the following conditions: (1) the input and output filters should present at least one output and input pin respectively, (2) the two pins through which the connection is made should support the same type of data stream, and (3) the minimum stream requirement dictated by at least one source and output filters with compatible media type. It receives as parameters the data types of the two filters, i.e., one audio/video format vector supported at the input of *InputFilter* and another one at the output of *OutputFilter*.

4.29 Listing 31 *AllowConnection* template class code

```
1  template <typename OutputFilter, typename
   InputFilter>
2  class AllowConnection {
3      typedef typename OutputFilter::output_types
   output_types;
4      typedef typename InputFilter::input_types
   input_types;
5
6  public:
```

```
7     enum { value = AllowConnectionImpl
  <mpl::size<output_types>::value>
8         ::template apply<output_types,
  input_types>::value };
9 };
```

Listing 32 partially presents the implementation of *AllowConnectionImpl* template structure that registers the result of connecting two filters into the enumeration variable value. This structure template is implemented by a specialized and a generic template, both receiving as template input parameter the number of elements of the *output_types* vector related to the *OutputFilter*. Also, both templates define a metafunction, *apply*, that takes the audio/video vectors supported by each filters as input parameters. For each element in the *output_types* vector it uses a boost MPL library metafunction named *contains* to check if the same element exists in the *input_type* vector. Any denied connection is verified and notified with an error message by the *CheckErrorOnConnection* template structure shown in Listing 33.

4.30 Listing 32 *AllowConnectionImpl* template structure code

```
1  template <int N> struct AllowConnectionImpl;
2
3  template <int N>
4  struct AllowConnectionImpl {
5      template <typename T, typename U>
6      struct apply {
7          typedef typename
   mpl::at<T,mpl::long_<N-1> >::type ftype;
8          enum { value =
   mpl::contains<U,ftype>::value —
9              AllowConnectionImpl<N-1>::template
   apply<T,U>::value };
10     };
11 };
12
13 template <>
14 struct AllowConnectionImpl<0> {
15     template <typename T, typename U>
16     struct apply {
17         enum { value = 0 };
18     };
19 };
```

4.31 Listing 33 *CheckErrorOnConnection* template structure code

```
1  // forward declaration
2  template <bool C, typename T, typename U>
   struct CheckErrorOnConnection;
3
4  // generic implementation
5  template <bool C, typename T, typename U>
6  struct CheckErrorOnConnection
7  {
8      static_assert(C, "cannot connect the two filters.");
9  };
10
11 // template specialization for filters v4l2 and xvideo
12 template <bool C>
13 struct CheckErrorOnConnection<C,CV4L2Source,
   CXVRenderer>
14 {
15     static_assert(C, "cannot connect v4l2 filter to
   xvideo filter.");
16 };
17
18 // specializations for the other filters combinations
19 ...
```

A denied connection occurrence is indicated by one of the three template parameters received by *CheckErrorOnConnection* template structure, i.e., the first one that has a boolean type. If denied, the value of *C* will be false as well as the input parameter of *static_assert* function and so, forcing a compilation error with our predefined message. For a more complete notification of a denied connection, specialized templates are defined, combining all participating filters.

The domain model will be concluded with the built-in intelligent connect feature of the filter graph composition and validation that is in charge of the following three tasks: (1) trying combinations of intermediate transform filters to satisfy the required matching between a pair of output and source pins, (2) specializing some partially configured filter that failed to be fully mapped to a model, and (3) avoiding code bloat by exploring any possible cross-silo synergy among similar filters. Possible code bloat may happen with a video surveillance system consisting of more than one filter graph using the same filter but with different configurations. In such case, several different classes will be instantiated, one for each configuration as illustrated in Figure 13.

To tackle the code bloat phenomenon, intelligent connect initially analyzes the type of filters and their settings for each pipelines and create a list of setting for any repeated filter among different pipelines but with different settings. Based on the collected settings, a unique class will be created, instantiated and shared among pipelines, as shown in Figure 14.

To conclude the domain engineering stage of the video surveillance systems development environment, a XSLT code transformer, shown in Listing 34, was implemented to generate a *Config.h* file from the configuration. This file will be used later by the offline partial evaluator C++

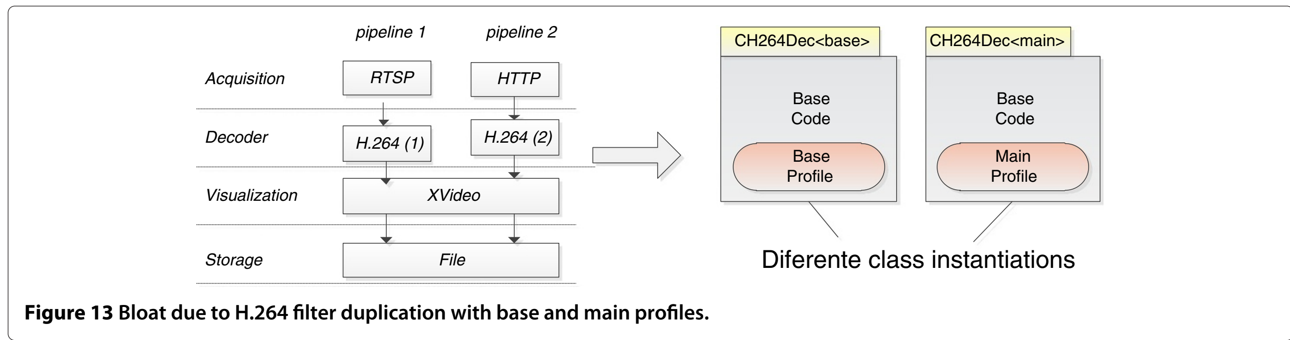

Figure 13 Bloat due to H.264 filter duplication with base and main profiles.

code generator that is in charge of the filter graph artifact template instantiation.

4.32 Listing 34 A snippet of the SDK built-in XSLT transformer

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xsl:stylesheet version="1.0"
3    xmlns:xsl=
     "http://www.w3.org/1999/XSL/Transform">
4     <xsl:template match="Configuration">
5  struct config {
6        <xsl:apply-templates select="FilterGraph" />
7
8     // list of filter graphs
9     typedef mpl::vector<
10       <xsl:for-each select="FilterGraph">
11       mpl::pair(fg<xsl:value-of select="position()"/>,
   c<xsl:value-of select="position()"/>)
12         <xsl:if test="last()-1 >= position()">
13           <xsl:text>,</xsl:text>
14         </xsl:if>
15       </xsl:for-each>
16     > fgs;
17 };
18 </xsl:template>
19 ...
20 </xsl:stylesheet>
```

5 Video surveillance application engineering

The passage from development for reuse to the development with reuse of video surveillance systems requires several steps surrounded by two main activities as shown by Figure 15: (1) establishing the communication channel between the stakeholders (IVV Automation customers) and system analysts and programmers to build a model family instance (i.e., a feature configuration and model templates), and then (2) generating the final code from the configured model family. At the end of a video surveillance system configuration process and before starting the code generation process, several concrete filters and filter graph will be encoded in XML and validated against the generic ones, and a mapping will be intensive and automatically established between features of the feature model and the model elements of the domain model.

Our MDD approach combines feature models and domain models on the artifact template class level encoded using C++ template metaprogramming. The mapping is automatically done at three level of granularity, i.e., at artifact template filter graph, at artifact template

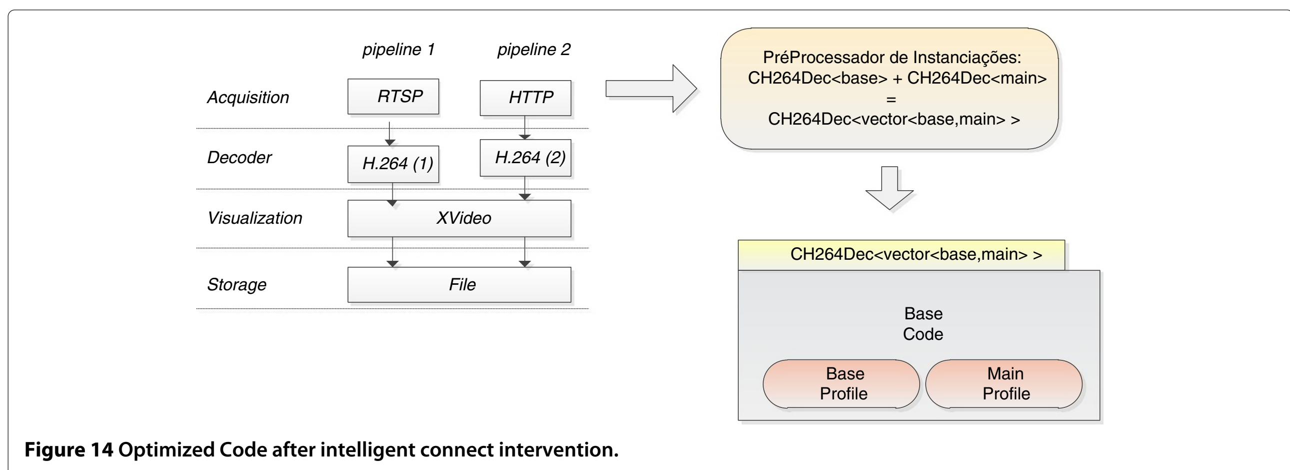

Figure 14 Optimized Code after intelligent connect intervention.

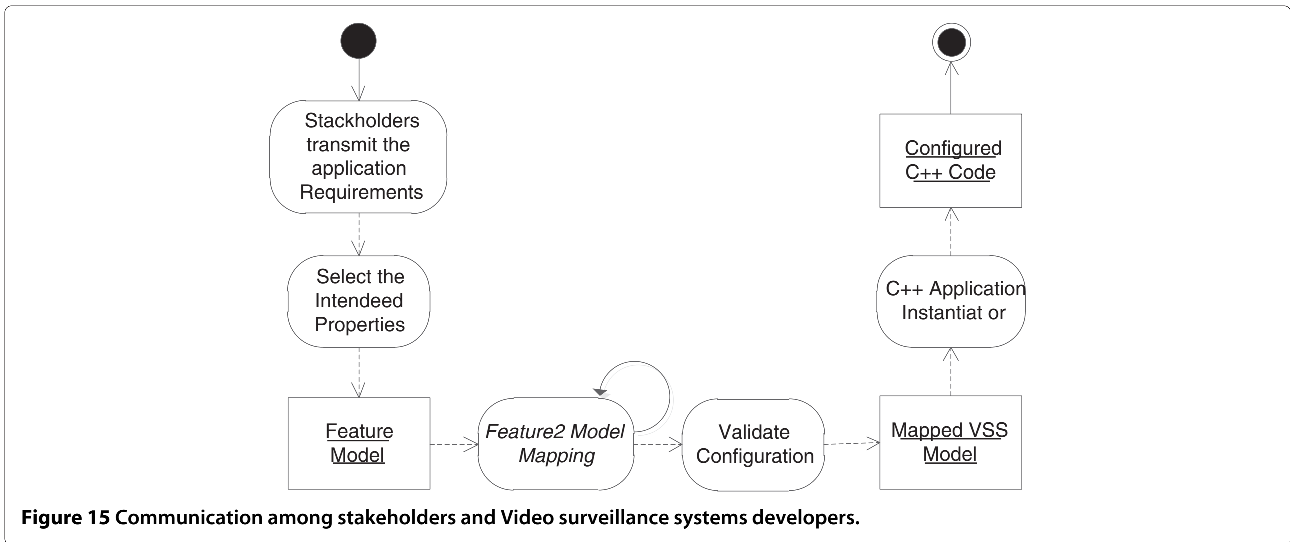

Figure 15 Communication among stakeholders and Video surveillance systems developers.

filter and internally to the artifact template filter at several variability points, such as attributes and methods when in the sync mode is selected or semi-automatically otherwise. Selected features of a filter will be successive collected till the moment that the filter can be evaluated by a meta-expression to a particular artifact template filter and the mapping realized. Any other selected features will be mapped to internal variability points or attributes of the previously mapped artifact template filter by meta-expressions or presence conditions, respectively, indicating whether they should be patched or removed from the specialized template instance. Mapping at template filter graph level will be triggered by a meta-expression when at list two filters are linked in a stream. Missing or unmapped feature in the domain model forces the extension of the domain model and so, be carefully analyzed. By now, the online partial evaluator based on C++ template metaprogramming solved it successfully, after refactoring some filter legacy code. Figure 16 shows a simple example of our *Feature2ModelMapping* activity.

Listings 35, 36 and 37 shows the second step of a two step video surveillance system generator, as the first one is a SDK built-in functionality. First, the XSLT transformer is fed with a video surveillance system configuration to generate the *Config.h* and then calling an offline partial evaluator based on C++ template metaprogramming (i.e., the *CApplicationInstantiator*), some of the previously artifact templates produced during domain engineering will

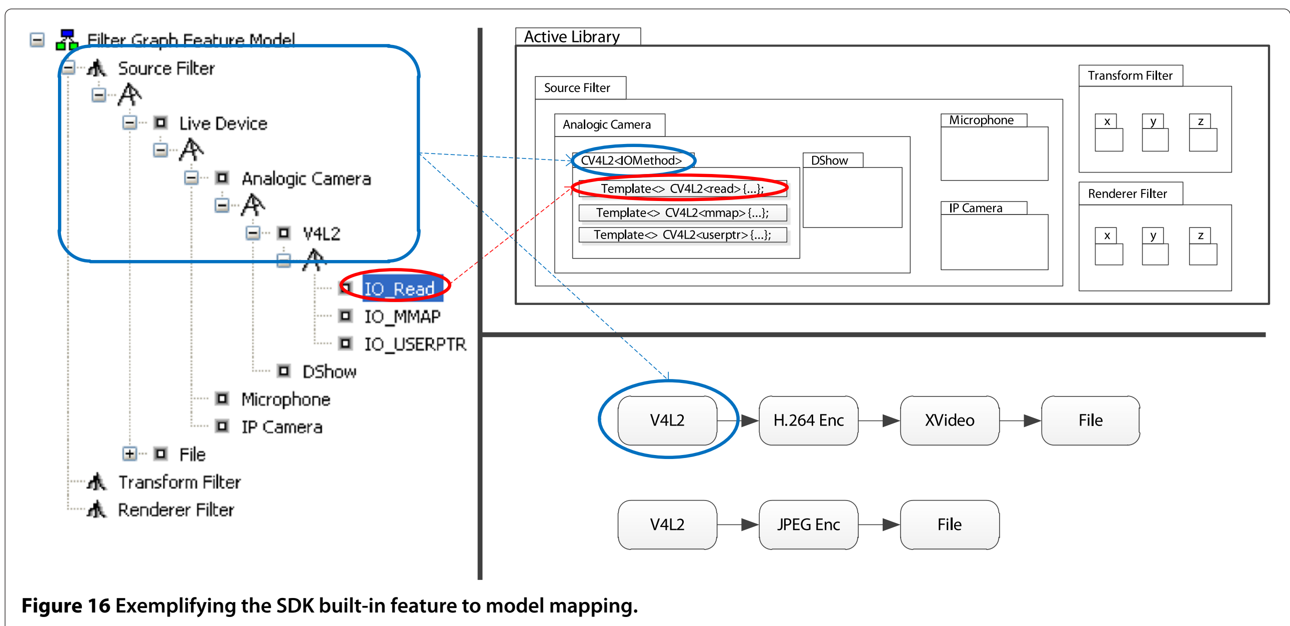

Figure 16 Exemplifying the SDK built-in feature to model mapping.

be customized into specialized application-specific components and linked together to construct the final video surveillance system that addresses the need of specific customers. Basically, the offline partial evaluator carries out the instantiation process at several levels of granularity by translating the video surveillance system configuration into an ordered sequence of finer instantiation operations which when executed will instantiate the new video surveillance system.

5.1 Listing 35 Offline Partial Evaluator: the *CApplicationInstantiator* class

```
1 template <typename Config>
2 class CApplicationInstantiator
3     : CApplicationInstantiatorImpl<
  IntelligentConnect<Config::fgs>::value,
  mpl::size<IntelligentConnect<Config::fgs>::value
  >> { };
```

5.2 Listing 36 A snippet of the Offline Partial Evaluator: the *CApplicationInstantiatorImpl* class

```
1  template <typename FilterTypes, int N> class
   CApplicationInstantiatorImpl;
2
3  template <typename Vector>
4  class CApplicationInstantiatorImpl<Vector, 1> {
5      typedef typename mpl::at<Vector,
   mpl::long_<0>>::type item0;
6      typedef typename mpl::first<item0>::type
   Filters0;
7      typedef typename mpl::second<item0>::type
   Connections0;
8
9      CFilterGraph<Filters0, mpl::size<Filters0>>
   _fg0;
10 public:
11     void Initialize() {
```

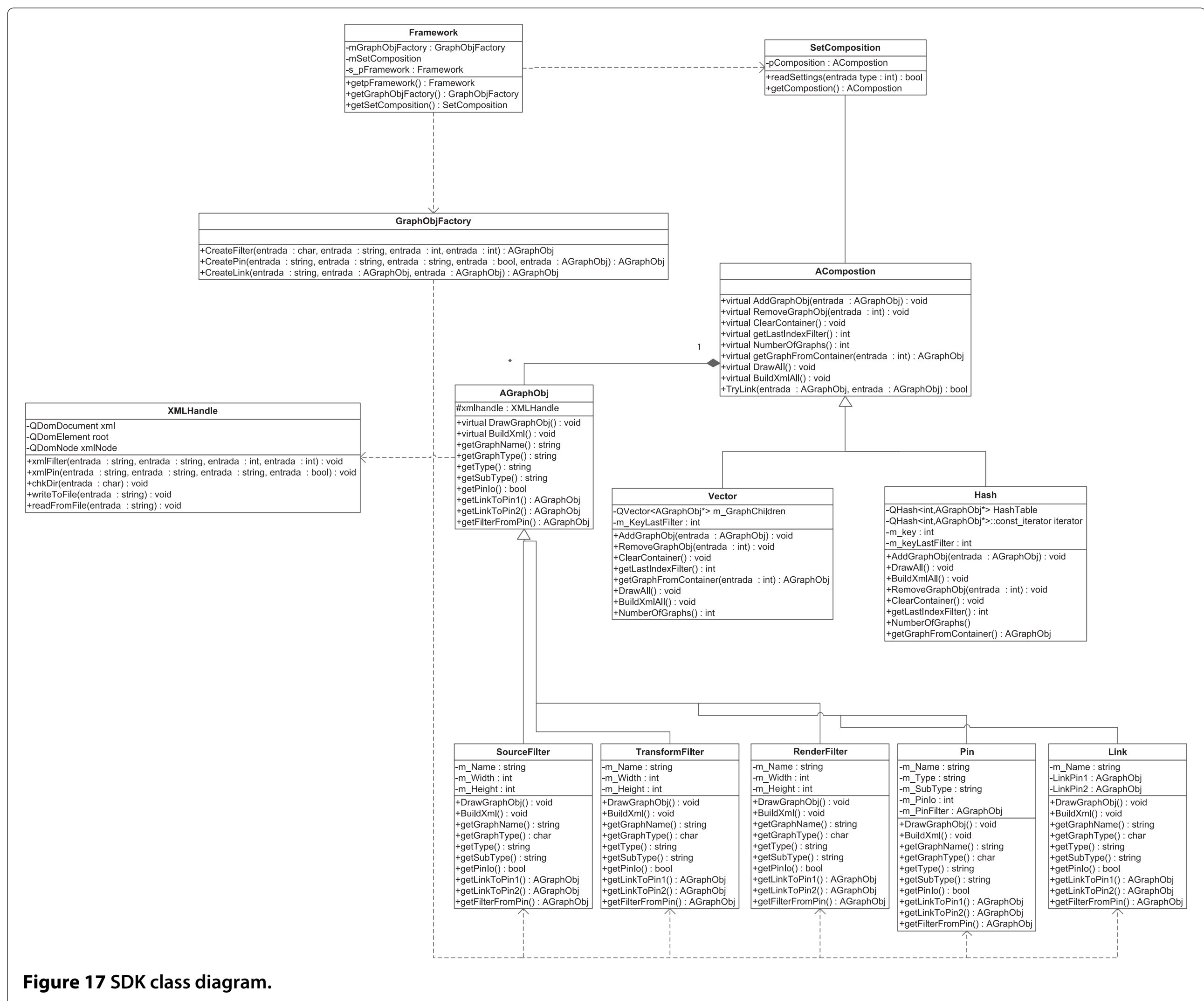

Figure 17 SDK class diagram.

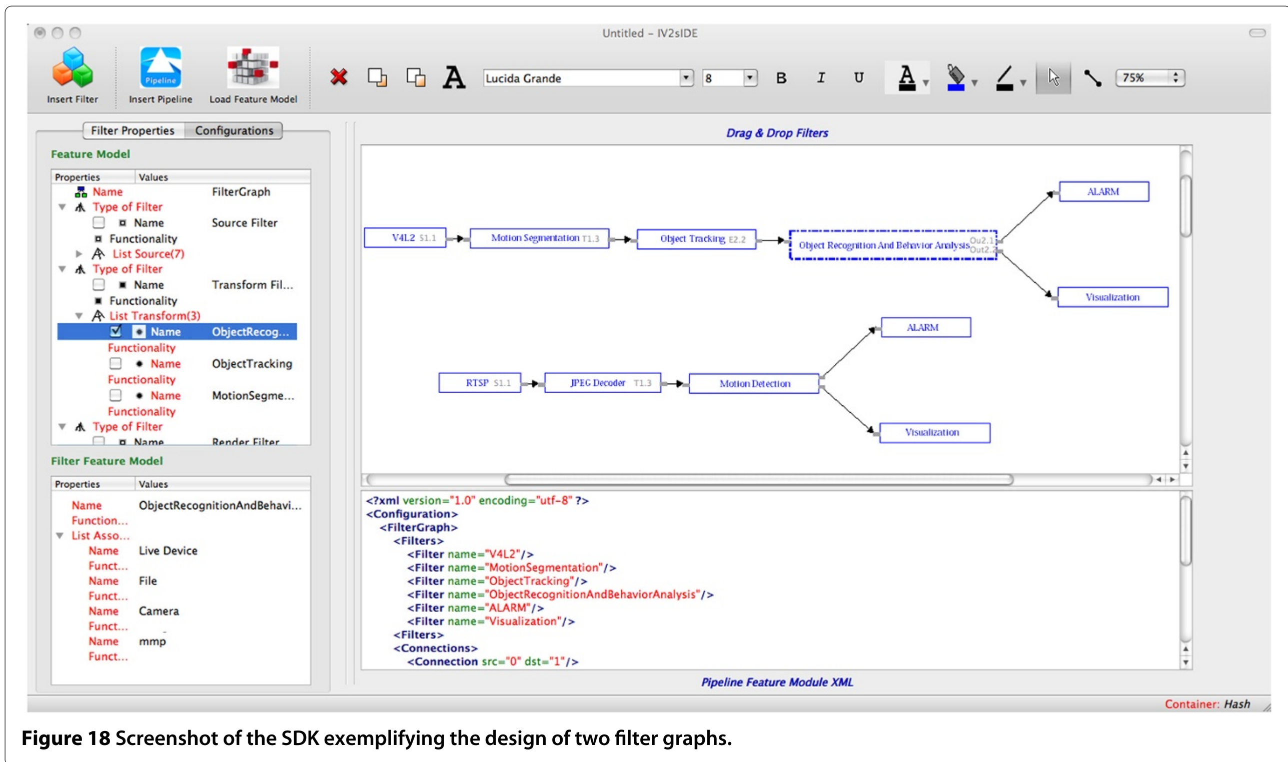

Figure 18 Screenshot of the SDK exemplifying the design of two filter graphs.

```
12          _fg0.ConnectFilters<Connections0>();
13      }
14  };
15
16  template <typename Vector>
17  class CApplicationInstantiatorImpl<Vector, 2> {
18      typedef typename mpl::at<Vector,
    mpl::long_<0>>::type item0;
19      typedef typename mpl::at<Vector,
    mpl::long_<1>>::type item1;
20      typedef typename mpl::first<item0>::type
    Filters0;
21      typedef typename mpl::second<item0>::type
    Connections0;
22      typedef typename mpl::first<item1>::type
    Filters1;
23      typedef typename mpl::second<item1>::type
    Connections1;
24
25      CFilterGraph<Filters0, mpl::size<Filters0>>
    _fg0;
26      CFilterGraph<Filters1, mpl::size<Filters1>>
    _fg1;
27  public:
28      void Initialize() {
29          _fg0.ConnectFilters<Connections0>();
30          _fg1.ConnectFilters<Connections1>();
31      }
32  };
33  ...
```

5.3 Listing 37 Offline Partial Evaluator: C++ final code generation

```
1  #include <config.h>
2
3  int main(int argc, char *argv[])
4  {
5      CApplicationInstantiator app<Config>;
```

Figure 19 Activities sequence of the early drowning detection at domestic swimming pools.

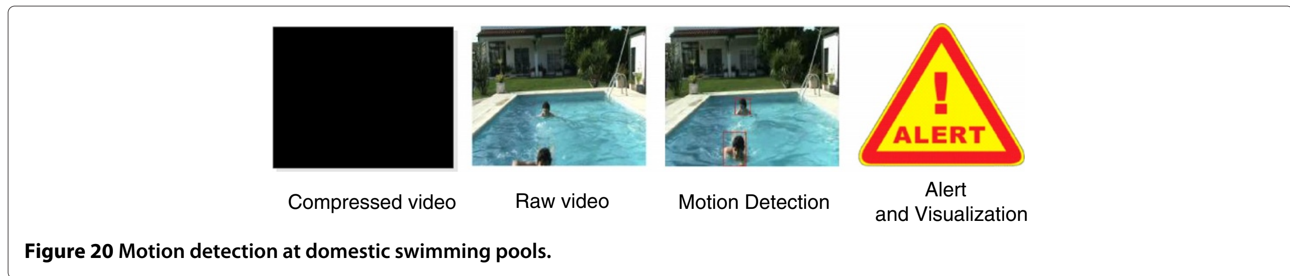

Figure 20 Motion detection at domestic swimming pools.

```
6     app.Initialize();
7
8     return 0;
9  }
```

6 Video surveillance product line SDK

A Qt-based video surveillance system SDK that allows graphical instantiation of a feature model configuration similar to DirectShow *graphEdit* was designed using several design patterns, such as singleton, strategy, factory and composite, as shown in Figure 17. After a configuration instantiation all the previously described steps will be executed till the generation of final C++ configured code for the new video surveillance system. Figure 17 presents a screenshot of the design of an early drowning detection at domestic swimming pools consisting of two pipelines: the first one with six filters:

(i) An analog input filter, V4L2, for image capture;
(ii) Three transform filters for motion segmentation, object tracking, and object recognition and analysis behavior;
(iii) Two output filter for alarm and visualization.

The second pipeline has the following five filters:

(i) A digital input filter, RTSP, that captures image frames from an IP camera using RTSP/RTP for transmission;
(ii) Two transformation filters for JPEG decoding and motion detection;
(iii) Two output filter for alarm and visualization.

Figure 18 illustrates two of several possible functionality scenarios supported by the generated system of Figures 19 and 20.

7 Methods

As previously mentioned, the video surveillance ecosystem is divided in two stages: the implementation domain, developed envisioning the system evolution, followed by the application engineering stage.

During the application engineering stage, a specific pipeline was created in order to meet the client imposed application requirements. In the pipeline creation process, several alternatives were explored, both at low levels (e.g., platform selection) and at high levels (e.g., filter selection according to context aware strategies.)

In this specific case, to make the comparison study between the two implementations, a video surveillance system based on DirectShow using C++, with virtual functions and inheritance, and a video surveillance system using the proposed framework were used on two kinds of machines. 1) First, a general purpose PC was used, an Intel Core 2 Duo CPU T9550 at 2.66GHz with 2,83 GB of RAM with Ubuntu 10.04 and secondly an embedded system based on a BeagleBoard-xM with an ARM Cortex-A8 MHz at 1GHz and extra memory with 512MB of low-power DDR RAM with Angstrom Embedded Linux.

For the two configurations, a pipeline composed by an RTSP source filter, a JPEG decoder transform filter and a File output filter was created. The pipeline was compiled with the GNU C++ compiler (g++) with the -O3 optimization option.

8 Conclusion and future work

To meet the embedded systems constraints and tackle the growing complexity of new video surveillance systems and time-to-market pressure, we abandoned our previous product-centric perspective and shifted to the multitude of products. In doing so, we promote the integration of MDD, SPL and generative technologies to create software for a portfolio of video surveillance systems in a pay-as-you-go fashion rather than an ex-nihilo development. The synergistic convergence of those technologies improved our ability to meet product quality demands while reducing the development effort and expanding the scale and scope of our video surveillance systems portfolio. One of the unique features of the proposed video surveillance systems design environment compared to existing ones, is its simplicity and higher integration level of mainstream tools and technologies such as, XML, XSLT, C++ template metaprogramming, generative programming, MDD, and SPL. Acher's et al. proposal didn't discuss the code generation process, i.e., it seems to us like an under development work. The proposed system supports the easy integration of external libraries that implement well-known standards and functionalities, e.g., ONVIF and

PSIA, developed as a filter that is integrated on the video surveillance system. The system was tested on a Beagle Board. The core processor of this embedded platform is the Texas OMAP that includes a DSP capable of decompressing HD frames in H.264 format, for which an encoding/decoding filter was developed. When the hardware system doesn't support SIMD instructions for video processing (MMX, SSE2, etc) or is not equipped with a DSP, then the video surveillance system performance will be degraded.

However, this project is still under development and next steps will be focused on: (1) automating the simulation and verification of the generated code, (2) automating the test cases generation and testing of the generated filters and video surveillance systems, (3) the integration of external modules, developed in other programming languages such as Python or Java (i.e., JNI-aware C++ wrapper), that implement video surveillance functionalities, and (4) study of the variability management of functionality related to playback application, e.g., playback, jump to absolute timestamp and find nearest I-Frame.

Endnote

[a]http://www.ivv-aut.com/.

Competing interests
The authors declare that they have no competing interests. This work was funded by the Science and Technology Foundation (FCT) and developed in cooperation with IVV Automation, but none pose any objection to the publication of results.

Acknowledgements
This work was funded through the Competitive Factors Operational Program - COMPETE and through national funds though the Science and Technology Foundation - FCT, within the project: FCOMP-01-0124-FEDER-022674. This work was developed in cooperation with IVV Automation; all support and means provided by the company is acknowledged.

Author details
[1]Department of Industrial Electronics, Centro Algoritmi, University of Minho, Braga, Portugal. [2]School of Engineering and Technology, Asian Institute of Technology, Khlong Luang, Thailand.

Received: 23 January 2012 Accepted: 5 June 2012
Published: 31 July 2012

References

1. L Marcenaro, L Marchesotti, C Regazzoni, in *Proceedings of the Fifth International Conference on Information Fusion*. vol. 2. A multi-resolution outdoor dual camera system for robust video-event metadata extraction, (Annapolis, MD, Washington DC Area, USA, 2002), pp. 1184–1189
2. M Acher, P Lahire, S Moisan, JP Rigault, in *Modeling in Software Engineering, 2009. ICSE Workshop on MISE '09*. Tackling high variability in video surveillance systems through a model transformation approach, (Vancouver, Canada, 2009), pp. 44–49
3. I Ahmad, Z He, M Liao, F Pereira, MT Sun, Special issue on video surveillance, IEEE Trans. Circ. Syst. Video Technol. **18**(8), 1001–1005 (2008)
4. Y Hongyu, X Lixia, X Feng, in *IEEE International Conference on Industrial Informatics*. Research on cluster remote video surveillance system, (Singapore, 2006), pp. 1171–1174
5. MD Pesce, *Programming Microsoft DirectShow for Digital Video and Television*. (Microsoft Press, Redmond, Washington, 2003)
6. PL Plauger, *Embedded C++: An Overview. Embedded Systems Programming*, (1997). [http://www.dsi.fceia.unr.edu.ar/downloads/informatica/info_II/c++/ec++.pdf]
7. S Meyers, Effective C++ in an embedded environment. [http://www.aristeia.com/c++-in-embedded.html]
8. D Saks, in *Embedded Systems Conference*. Reducing run-time overhead in C++ programs, (San Francisco, USA, p. 2002
9. 9. Technical Report on C++ Performance. [www2.research.att.com/bs/performanceTR.pdf]
10. I McRitchie, TJ Brown, ITA Spence, Managing component variability within embedded software product lines via transformational code generation, Softw. Product Family Eng. **3014/2004**, 98–110 (2004)
11. A Kleppe, J Warmer, W Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, Boston, 2003)
12. K Czarnecki, U Eisenecker, *Generative Programming: Methods, Tools, and Applications*. (Addison-Wesley Professional, Boston, 2003)
13. R Anisko, Towards automatic partial evaluation for the C++ language, in *CSI Seminar* (2002)
14. D Lohmann, W Hofer, W Schroder-Preikschat, O Spinczyk, in *AOSD '11 Proceedings of the tenth international conference on Aspect-oriented software development*. Aspect-aware operating system development, (Porto de Galinhas, Pernambuco, Brasil, 2011), pp. 69–80
15. V Sarinho, A Apolinario, in *SBGAMES '09 Proceedings of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment*. A generative programming approach for game development, (Rio de Janeiro, Brazil, 2009), pp. 83–92
16. T Buchmann, A Dotor, in *Pieter van Gorp, Hrsg., Proceedings of the 7th International Fujaba Days*. Mapping features to domain models in Fujaba, (Seiten 20–24, Eindhoven, The Netherlands, 2009)
17. AMPLE: Aspect-Oriented, Model-Driven, Product Line Engineering. [http://cordis.europa.eu/search/index.cfm?fuseaction=proj.document&PJ_RCN=8724342]
18. H Wada, EMM Babu, A Malinowski, J Suzuki, K Oba, in *IASTED Int'l Conf. on Software Engineering and Applications*. Design and implementation of the matilda distributed UML virtual machine, (Dallas, TX, USA), p. 2006
19. K Czarnecki, U Eisenecker, in *Proceedings of the European Software Engineering Conference*. Components and generative programming, (Toulouse, France, 1999), pp. 2–19
20. IDA: About. [http://www.hex-rays.com/products/ida/index.shtml]
21. K Kang, S Cohen, J Hess, W Nowak, S Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study,* (CMU/SEI- 90-TR-21). (Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990). [http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm]
22. V Cechticky, P Chevalley, A Pasetti, W Schaufelberger, in *GPCE '03 Proceedings of the 2nd international conference on Generative programming and component engineering*. A generative approach to framework instantiation, (Erfurt, Germany, 2003), pp. 267–286
23. K Czarnecki, M Antkiewicz, in *ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'05)*. Mapping features to models: a template approach based on superimposed variants, (Tallinn, Estonia, 2005), pp. 422–437
24. A Gurtovoy, D Abrahams, The boost MPL library. [http://www.boost.org/doc/libs/1_48_0/libs/mpl/doc/index.html]
25. D Abrahams, A Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. (Addison-Wesley Professional, Boston, 2004)

doi:10.1186/1687-3963-2012-7
Cite this article as: Cardoso *et al.*: **A generative-oriented model-driven design environment for customizable video surveillance systems.** *EURASIP Journal on Embedded Systems* 2012 **2012**:7.