Abstract—The interoperability of IP video equipment is a critical problem for surveillance systems and other video application developers. ONVIF is one of the two specifications addressing the standardization of networked devices interface, and it is based on SOAP. This paper addresses the development of an ONVIF library to develop clients of video cameras. We address the choice of a web services toolkit, and how to use the selected toolkit to develop a basic library. From that, we discuss the implementation of features that are important challenges for developers and, therefore, opportunities for providing higher-level functionalities. The proposed solutions result from the implementation of library that has been validated in industry.

Index Terms—ONVIF, SOAP, WS-Discovery, WS-Security, web services, NVT.

I. INTRODUCTION

Internet Protocol (IP) video equipment differs in a variety of ways, from provided features, to network configuration, user management, video encoding/compression schemes, supported network protocols, etc. The software interface offered by manufacturers to configure and use this diversity of features also varies. In this scenario, interoperability becomes a serious challenge, with numerous shortcomings for everyone (end users, integrators, and manufacturers). To tackle the problem, two industry groups – the Open Network Video Interface Forum (ONVIF) and the Physical Security Interoperability Alliance (PSIA) – were formed, with the goal of standardizing IP video surveillance. Both groups came out with specifications, both based on web services.

Web Services (WS) enable machine interoperability over a network by defining a standard way to describe service operations, data format and network protocol. Usually they are based on open and well-established standards such as HTTP for information transport and XML for data serialization. Two main approaches exist: REST-compliant and SOAP-based WSs.

PSIA follows the former [1], while ONVIF follows the latter [2]. Besides that, they have two main differences: PSIA has a broader scope, addressing systems-level concerns like access control, while ONVIF is more focused on the device-level functionalities; ONVIF is built on WS standards and specifies more constraints than PSIA. For example, for events the former uses WS-BaseNotification [3], while the latter defines only the format of message headers but not the bodies. The current PSIA vagueness means more complexity for implementing interoperability.

Our work addresses ONVIF, namely the development of a client library. ONVIF adopts a significant number of web standards (e.g., WSDL, SOAP, WS-Discovery [4]) and consists of several specifications. Consequently, it is not trivial to implement, and demands time and effort to get acquainted with all specifications. Moreover, some ONVIF specifications have a significant level of complexity. Overall, there are several opportunities to develop a library that can reduce the difficult and effort necessary to develop client applications.

To our knowledge, ONVIF Device Manager [5] is the only non-proprietary library available. It is an open source C# library to manage Network Video Transmitter (NVT) devices that comes with a demo application. Moreover, and more important, industrial applications seek to increase efficiency, to raise competitiveness and mark a position on markets. So, computational performance and resource usage is very important and the use of non-interpreted programming languages has advantages in this matter.

In the literature, there are few works involving ONVIF. The more focused ones [6,7] describe parts of ONVIF and particular applications. They do not address the problem of implementing an ONIF library that provides generic functionality to help to develop NVT client applications.

In this paper, we describe the process of developing a client library for NVTs using C/C++. We present our experience in...
developing a library for an industry partner that integrates IP cameras to produce a complete line of surveillance equipment. More concretely, we (1) explore the choice of a WS development toolkit implementing the standards used by ONVIF, (2) distill ONVIF’s intricate parts, (3) systematize the challenges involved in the library development, and (4) offer generic solutions or guidelines to the problems faced.

In the next section ONVIF is introduced. Section III presents available toolkits and the proposed choice. The development of an ONVIF library based on the chosen toolkit is described in section IV. In section V, we address important challenges and details on how to add higher-level functionality to the library. Finally, conclusions and future work are pointed out in section VI.

II. ONVIF OVERVIEW

ONVIF functions are defined as SOAP operations. Core specification [2], contextualizes the usage of WS-Discovery, WS-Security [8] and WS-BaseNotification, and defines Device Management (DM) and Event service. Other specifications define a single service. ONVIF devices are classified in types – NVT, NV Display, NV Storage and NV Analytics – for which a set of mandatory (M), mandatory if it has a related feature (C), and optional (O) services are defined. A NVT device has the following set of services: M = {Device Management, Event, Media, Streaming, Device IO}, C = {PTZ}, and O = {Imaging, Video Analytics}.

Services’ operations are protected by user authentication and a security policy. Authentication credentials should be provided at either the transport-level (HTTP), using Digest Access Authentication [9], or the message-level (SOAP), using one of token profiles defined by WS-Security. Servers shall support at least the WS-Security UsernameToken Profile [10], which requires clients to send user name of an existing account and respective password digest.

In what concerns device security policy, (1) each user account has associated one of four privilege levels – Administrator, Operator, User, Anonymous, and (2) each service operation has a predefined access-level – e.g., PRE_AUTH, READ_SYSTEM, WRITE_SYSTEM. The security policy is a table that establishes the relationships between privilege-levels and access-levels. Both the security policy and the user privilege-level can be changed by a client.

ONVIF enables the development of automation systems based on NVTs, wherein the controller is a client. The input of such systems can be both video and digital inputs, and the control can be made through NVT relay outputs or any other devices. The detection of actuation scenarios can be made by processing video streams at either ONVIF clients or NVTs supporting the optional Video Analytics service. NVT inputs and outputs command is made through the mandatory Device IO service. Asynchronous real-time control can be implemented by using the Event service. This service enables the subscription of input and Video Analytics events that are automatically sent to the client controller when detected by NVTs (through Basic Notification Interface [8,2]). Events can also be received on request (using Real-time Pull-point Notification Interface [2]) and through metadata streaming (Notification Streaming Interface [2]).

III. WEB SERVICE TOOLKITS

WS toolkits help developers to build applications, by providing tools that automate important tasks and APIs that implement useful features. Usually they offer a WSDL compiler to generate client stubs and server skeletons code from WSDL documents that define services. The generated Remote Procedure Call (RPC) code includes messages parsing with validity checking and type-safe (de)serialization of function parameters’ data, hiding from the user irrelevant WSDL, SOAP and XML details. Therefore, they reduce development time and enable programmers to concentrate on application logic.

Among toolkits that support SOAP, generate C/C++ code and are freely available, we describe the more relevant. A complete review of available toolkits supporting any kind of web-services is out of the scope of this article. This section makes a brief comparison of toolkits from the perspective of implementing ONVIF services, based on experiments.

A. gSOAP

gSOAP [11] is an open source toolkit for developing SOAP/XML web services and XML applications, and it is available for many platforms, including Linux, Windows and MacOS. Currently, it supports a wide variety of WS standards and protocols, including several useful for ONVIF. For example, gSOAP provides C/C++ plugins supporting WS-Addressing, implementation of clients, servers, and discovery proxies according to WS-Discovery, and security models defined by WS-Security.

gSOAP contains two compilers, wsdl2h and soapcpp2, which are illustrated in Fig. 1. In the upper part and in black, it shows the development of a client application for an existing service from its WSDL document, wherein wsdl2h parses WSDL and XML schema (XSD) definitions converting them into annotated C/C++ code – the gSOAP Remote Procedure Specification (gRPS) file. This is a header file containing functions declarations and data structure definitions, from which soapcpp2 generates client RPC stubs. In the lower part and in gray, Fig. 1 shows the development of a WS application from a gRPS file describing the interface of existing code (to be ex-
posed as a WS). From it soapcpp2 generates client stubs, server skeletons and WSDL document of the service.

B. Apache Axis2/C

Apache Axis2/C project includes toolkits for C, C++ and Java web services. Axis2/C (http://axis.apache.org/axis2/c/core) is available for Linux and Windows platforms and, similarly to gSOAP, supports a number of WS specifications. It contains the WSDL2C tool to generate C code. The latest version (1.6.0) is not updated for more than 3 years and it has a limitation on the processing of WSDL files: it requires the definition of at least one service to generate stubs and skeletons. The WSDL files provided by ONVIF do not have any service defined, but only binding elements (that define message format, protocol details and available operations). Consequently, to be used with ONVIF it is necessary to manually modify the standard WSDL files, to include a service with endpoints (ports) for all bindings. After that, WSDL2C generates four files for each operation, which is not user-friendly and thus complicates development. For example, the DM service alone has 82 operations.

C. WSF/C

WSO2 frameworks (http://wso2.com/products/web-services-framework) add tools to the Apache Axis2 toolkit, and support C, C++, Java, PHP and other languages. WSF/C (and the WSF/C++ wrapper) uses Axis2/C engine and the Axis2Java tool to generate C code. It does not provide (as of version 2.0.0) a library to implement WS-Discovery. To use WSF/C in platforms other than Windows, it is necessary to compile the source code. The documentation does not describe how to generate stubs for a client application from WSDL files. The code generation tool needs Axis2/Java, which has to be installed separately and configured; this is documented only in a readme file deep down in the directory tree.

D. Staff

Staff (http://code.google.com/p/staff) is based on Apache Axis2/C, but supports more platforms, namely MacOS and FreeBSD. Similarly to gSOAP it also uses a header file that defines the service interface. From WSDL files, the staff_codegen can generate C++, AJAX and J2ME code for clients, and C++ server skeletons. It can also generate WSDL documents from service interface header files, and project files for Microsoft Visual Studio and Qt Creator.

We used the last available version (v2.0.0-a1-r686) to generate stubs for DM service (using devicemgmt.wsd file). Similarly to the toolkit it is based on, it requires the modification by hand of standard ONVIF WSDL files, otherwise only data definitions are generated. From WSDL files with a service defined, it generates stub definitions, as well as a main program which demonstrates how to invoke all stubs (including the necessary variable declarations). In contrast with Apache Axis2/C, staff_codegen generates three files for each namespace in WSDL files, in two steps: first, a service interface header file, and afterwards, two (.h and .cpp) stub/skeleton files (i.e., the proxies).

E. Chosen toolkit

More toolkits can be found, but these have important limitations and errors in some cases. Taking in consideration both the features and the experience in using several toolkits, our choice was gSOAP mainly due to: reduced number of files and consequent decrease of complexity (hiding information about underlying web-services); compatibility with standard ONVIF WSDL files; very complete set of documentation (with many code samples and examples, covering a diversity of aspects); support of a wide range of WS specifications. Moreover, it is frequently updated, has a valuable support forum, and it is easy to use (including the processes of installation, code generation and integration with client code). Finally, it is used by developers worldwide and has been tested over many years.

IV. LIBRARY DEVELOPMENT APPROACH

A. Using gSOAP

Developing a library for NVTs with gSOAP basically consists in downloading the WSDL files of involved services from ONVIF website and generating client stubs according to the description given in III.A.

For example, the stub function that is generated for the SetSystemDateAndTime operation (of DM service) is shown in Fig. 2. Note that besides the last two arguments, which contain the request and response SOAP bodies, respectively, there are three more parameters. The first one is a pointer to the exclusive runtime context of each gSOAP method, which includes, for example, SOAP header fields for both request (e.g., authentication credentials and communication parameters) and response (e.g., error messages). The second parameter is the address of the device. The third one is the optional SOAP action; if NULL it is filled with the appropriate value by default. These five parameters form a pattern common to all gSOAP stubs.

ONVIF relies on several standard WSs whose definition does not include concrete operation bindings, nor services and ports. This is the case of WS-Addressing, WS-Discovery and WS-Security. Consequently, gSOAP does not generate stubs for them, but they are supported by plugins.

B. Functionality-level

A straightforward approach to develop the library would be to offer an API consisting of client stubs for ONVIF NVT services, and WS-Addressing, WS-Discovery and WS-Security functions of gSOAP’s plugins. However, such a library would miss many opportunities to provide a higher-level of functionality and simplify development of client applications. Although ONVIF stubs, like the one in Fig. 2, take care of

```c
int soap_call__device__SetSystemDateAndTime(
    struct soap *soap,
    const char *soap_endpoint,
    const char *soap_action,
    struct_device__SetSystemDateAndTime *device__SetSystemDateAndTime,
    struct_device__SetSystemDateAndTimeResponse *device__SetSystemDateAndTimeResponse);
```

Fig. 2. gSOAP stub generated from ONVIF.
SOAP protocol, the developer still has to provide all of its arguments. The request argument can have mandatory and optional fields. Moreover, stubs do not hide details of ONVIF specifications, forcing the developer to deal with them. Plugins’ APIs also expose details of the respective services. Finally, the developer also has to integrate plugin functions with stubs.

In the following section we explore the abovementioned issues in detail and provide solutions to implement a library that raises abstraction-level and simplifies development of NVT clients.

V. PROBLEMS AND SOLUTIONS

An issue that cross-cuts all services is that stubs do not validate request contents before sending them to the network. Therefore, if a mandatory field of a request is missing, it is still sent, and will inevitably result in an error. For example, if “soap_call___device__SetSystemDateAndTime” (see Fig. 2) is called with a 5th parameter that includes only the hour (and forgets the date), the request message is sent. The response contains an ONVIF fault (“InvalidArgs”). The library can avoid this by checking if mandatory fields are present in the request arguments and thus prevent unnecessary network traffic and increase performance.

A. Device Discovery

The first thing a client wants to do is to find NVTs on the network. gSOAP offers a plugin supporting both version 1.0 and 1.1 of WS-Discovery. The plugin API provides functions to send all service messages (Probe and Resolve for clients, and Hello, Bye, ProbeMatch and ResolveMatch for servers).

To use the plugin for ONVIF, it is necessary to add to library’s gRPS header file the line: #import “wsdd10.h” (“wsdd.h” is not compatible). This line makes soapcpp2 compiler generate the functions (stubs) that the plugin API uses to send WS-Discovery messages. Code that uses the plugin must include “wsddapi.h” (from “plugin” folder), and compile the respective .c and other modules it depends on.

Consider the (probably most common) case wherein an ONVIF client wants to find all NVT devices in its local network and it can connect to the network (or turned-on) at any time. Figure 3 lists a code example to implement that search. To find devices that have been connected before, a client must send a Probe message using “soap_wsdd_Probe” function. The programmer has to provide the following arguments, in order:

1. A gSOAP context,
2. ad-hoc (multicast) discovery mode to
3. target services,
4. multicast group address,
5. unique message identifier (previously built using the WS-Addressing plugin),
6. no address to relay matches (i.e., they will be delivered to the probing client),
7. ONVIF device type,
8. scopes and
9. respective matching rule.

The last three optional parameters define conditions that devices must meet, thus narrowing search results.

When ad-hoc mode is used, Probes (and other) are one-way SOAP messages. Consequently, the client must afterwards wait for arrival of ProbeMatch messages sent by devices. This is done by using “soap_wsdd_listen” function, which receives any discovery message and calls a handler for each one until the timeout value is reached. There are six handlers, one for each kind of message, and all must be implemented. Empty (or dummy) implementations can be provided for handlers whose messages are to be ignored. This is what happens with all handlers of our example, except the one for ProbeMatches, whose definition is shown in Fig. 4.

When a ProbeMatch is received, “listen” function calls “wsdd_event_ProbeMatches” handler, giving client the opportunity to process message data. Since ProbeMatches are sent asynchronously, handler action should be minimal: (1) check if they are related to the previously sent probe (using the “RelatesTo” argument) and, (2) if so, save received matches (the last argument) in the user part of context (“ctx.user”). Only after “listen” is finished, should the client make further processing, namely find and save the address of DM service of each match.

The functionality of this example can be implemented by a function that has only a few parameters. The context is common to all gSOAP functions, and can be handled internally. The same applies to the message identifier of “soap_wsdd_Probe”. The 2nd to 4th and 7th parameters are constant. The same applies to the 6th, if no message relay is desired. To maximize compatibility, 9th parameter should be null, which defaults to the matching rule all ONVIF devices must support (RFC 3986). Consequently, the only parameter that matters is the list of scopes, and the search of all NVTs can be implemented by a function with the following prototype:

```
int find_NVTs(const char *Scopes, DeviceInfo **list);
```

It has only an input parameter, and the second one is the output list with all matched DM service addresses and other ONVIF defined info, such as location, hardware, etc.

This function is obviously much easier to use, than having to deal directly with the discovery API. The reader may argue that, however, it narrows discovery options. If managed mode through a discovery proxy is desired, it can be supported by

```
void wsd_d_event_ProbeMatches(struct soap *soap, 
unsigned int InstanceID, const char *SequenceID, 
unsigned int MessageNumber, const char *MessageID, 
const char *RelatesTo, 
struct wsdd__ProbeMatchesType *matches); 
```

Fig. 4. Prototype of ProbeMatch messages handler.

Fig. 3. Code discovering all NVTs in a network.
adding a function with one more parameter to the library API, which would abstract the 2nd to 4th parameters of “soap_wsd Probe”. This parameter would be address of the discovery proxy or, when null, would mean ad-hoc mode. In the latter case, if a discovery proxy is found, the signal to switch to managed mode can be provided through the return value or a special value in the list of devices found. If the library intends to support any ONVIF device, then the device type (7th) parameter would have to be offered. Nevertheless, such a function, which would not limit any discovery options, would have only 4 parameters.

B. Authentication

With exception of a few functions of DM, all ONVIF functions require authentication. Clients must provide authentication data according to WS-Security: in the SOAP header, which is part of gSOAP context. The WS-Security plugin of gSOAP provides functions that calculate and (de)serialize authentication data out of/into the context. To use them, it is necessary to import “wsse.h” in the grPS file, include “wsseapi.h” in the code and compile the respective modules, similarly to what is needed for WS-Discovery.

As mentioned in section II, the security model that is common to all ONVIF devices is UsernameTokenProfile. Its password digest is the Base64 encoding of SHA-1 value that results from the concatenation of three parameters: nonce, creation timestamp and password. Consequently, client applications must write authentication data to the context, using “soap_wsse_addUsernameTokenDigest” function of the plugin, prior to each stub call. Figure 5 shows the example of a client that configures a device to synchronize via Network Time Protocol (NTP). First the stub for setting NTP server options is used, and then synchronization is activated using “SetSystemDateTimeAndTime” stub.

When a device is being operated, its address (2nd parameter) and user credentials (name and password) are the same for all stub calls. Therefore, the library can abstract these parameters and automatically provide authentication for stubs. To implement such functionality the library needs to: (1) implement a function – “setDeviceAndUser” – that (globally) stores service address and user credentials, to be used in subsequent stub calls; and, (2) wrap each stub in a function that uses previously stored data to fill the context with authentication. The result is shown in Fig. 6, which makes evident that the approach greatly simplifies the developer job. Such a library has state – the current selected device address and user credentials. The developer only needs to call “setDeviceAndUser” when switching the operated device. Furthermore, the wrapping of ONVIF functions that have an empty response also eliminates the respective parameter of the stub (the last one in the example).

Since each ONVIF device provides several services with different addresses, “setDeviceAndUser” must store the address of all services. These are obtained through the “getCapabilities” stub of DM service, in Fig. 7. This gives to each library function the opportunity to test if its service is available before sending any request, and thus save time and network bandwidth.

C. Media

The Media service specification is particularly complex, because it defines several concepts – profiles, sources, configuration entities (CE), and options – and relationships between them (see Fig. 8). A media entity represents a feature that devices can have, and ONVIF specifies nine of them (Video Source, Video Encoder, Audio Source, Audio Encoder, Audio Output, Audio Decoder, PTZ, Video Analytics and Metadata). For each entity zero or more configurations can exist. Configurations cannot be created neither deleted. Their parameters’ values (hereafter called CE value, for simplicity) can be changed according to the respective available options (except for Video Analytics, which does not have procurable options). A media profile is composed by at most one configuration of each entity, according to the device resources.

The dependencies between all CE and respective sources are described in Fig. 9. For instance, a VideoEncoderConfiguration requires a VideoSourceConfiguration, which in turn is associated with a VideoSource. Each PTZConfiguration concerns a PTZNode, being both defined as part of the PTZ service.

NVT devices come from factory with at least one profile. Consider the case of a client that wants to create a new profile, or entirely customize a predefined one. After verifying which optional capabilities are available, it needs to find the ranges, bounds, and sets of values that are allowed for each CE. This is accomplished by calling at least 7 stubs of the form

\[
\text{r = soap_call___media__Get<CE>Options}(
\&ctx, \text{media_addr}, \text{NULL}, \text{NULL}, \text{NULL}, \text{&request}, \text{&response})
\]

one for each white-filled CE in Fig. 9, wherein \(<CE>\) is the respective name. An additional and similar stub from the PTZ service will be used to get PTZConfiguration options. The parameters follow the pattern described in section IV-A, and action can be null. The request data consists only of optional tokens of a profile and a configuration. When null, the returned options are applicable to all configurations and profiles.

The library should include a function that automates options retrieval by (1) picking the previously saved media ser-
vice address, (2) taking care of details (context, authentication and action), (3) providing request data to obtain generally applicable options, (4) calling only the stubs of available capabilities (avoiding non-supported optional services) and (5) collecting all options from responses. Such a function has only one output parameter – the options for all supported configurations – and calling it would be as simple as

```c
r = getAllMediaOptions(&configs_options);
```

To create a new profile ONVIF requires the client to (1) create an empty profile (using “CreateProfile” stub), (2) obtain the desired configurations (using “get<CE>s” stub to obtain all CEs of each kind and selecting one), (3) set new values on the selected CEs (using “SetConfiguration” stub) and (4) add them to the new profile (using “AddConfiguration” stub). To customize an existing profile the client must first obtains its CEs (using “getProfile” stub and finding them on the response data) and then carry out steps similar to the abovementioned: to change parameters of existing CEs, it performs step 3; to add missing CEs, it performs step 2 for the respective kinds, and then steps 3 and 4; to remove undesirable CEs, it uses “RemoveConfiguration” stub.

To simplify the Media service interface, the library can provide a “setProfile” function that provides both the functionalities of adding a new profile and changing an existing one, in a uniform way. This function abstracts all aforesaid stubs and the relationships of Fig. 9. The “setProfile” function only needs to receive as arguments a profile token, and the CEs’ values for it. All stubs’ request data parameters can be derived from that information, and intermediate responses, according to what is explained above. The first 2 parameters have already been explained in previous sections, and default actions (provided by stubs) are used. The prototype can be

```c
int setProfile(char* profile_token, struct CEsValues* values);
```

If a profile with the received token does not exist, it is created; otherwise its CEs will be changed/updated. The 2nd parameter is a library defined structure capable of holding values of all possible CEs, but not the CEs themselves (i.e., not their tokens). When the value of a given CE is absent, it means the CE will not be part of the profile (i.e., be removed or not added). Since CEs can be shared among profiles, changing one profile can propagate to others. The library can avoid this by preventing CE sharing: “setProfile” should manage the use-Count property of CEs as their added to and removed from profiles, and only add those that are free (useCount = 0). If a free CE does not exist, than an error must be issued.

**D. Error Management**

In case of failure, functions that perform several operations, like “setProfile”, must leave the device as it was before their execution, i.e. they must behave as if they were atomic. Therefore, they have to store data that enables to restore the original state. For example, “setProfile” needs to save the token of a created profile or a copy of a to-be-changed profile.

Error management is an important aspect of the library also because errors can occur at different levels: HTTP, SOAP, ONVIF and library. For example, a SOAP error occurs when a message has a wrong name or field, while an ONVIF error happens when trying to obtain the streaming URI for a profile without a VideoSourceConfiguration. Library errors identify situations that occur in its functions and violate assumptions or rules, as exemplified at the end of previous section. With several error sources it is necessary to integrate all error values while allowing source identification. Hence, error values should be divided in four separate bit-fields, according to the sources/levels mentioned above.

Furthermore, the library should offer functions that give access to generic error descriptions, for example like the standard C “perror” and “strerror”. However, some ONVIF errors include more detailed descriptions. Therefore, two error description levels should be introduced, opening door to provide also details about library functions’ errors. Detailed descriptions can provide concrete feedback on input arguments that originated the error, or point to actions that prevent the error from happening. This way, the error mechanism becomes a more useful tool.

Such functionality can be implemented by simply writing error descriptions to a global structure when they occur, and cleaning that structure in case of success. Applications can then access error information by using “getError” and “getErrorDetails” functions, whenever the library returns with an error code. HTTP and SOAP error descriptions are provided by gSOAP (in the context). ONVIF error descriptions are defined in specifications and must be implemented by the library, either in a centralized internal table indexed by the error codes, or spread throughout the code. This error reporting mechanism is adequate to provide feedback on applications’ user-interface.

The library can also be instrumented with debugging code that prints errors to standard error output, which is important for library evolution and maintenance. The debugging mechanism can be flexible and readable way using a C pre-processor macro and a command-line argument. Each usage of the macro is a single line of code that defines a debug message and its level. The command-line argument defines at application load time the threshold for printing debug messages. The macro will only print messages whose level is higher than the threshold,
and debug lines can be removed from release binaries by a simple compilation switch.

VI. RESULTS

We have developed a library for NVT clients that uses gSOAP and implements the solutions proposed in the previous section. It was developed and tested using several IP cameras with ONVIF support, such as Vivotek IP8151, Lilin IPD112ESX and Dynacolor NA222. All library features were tested successfully, although we faced minor problems in cameras’ implementation of ONVIF. These problems are considered as normal, since ONVIF specifications are recent and evolving. Consequently, manufacturers are still making progresses to implement flawlessly all ONVIF services.

The library has already been recently delivered to the industry partner and is being easily integrated, replacing legacy code. The library offers functionalities designed to be generically applicable, and the only request we had was the addition of a function that addresses specific needs of the partner: provide resolutions of all codecs supported by a device (whether or not included in VideoEncoderConfigurations of existing profiles), and all existing streaming URIs. This function is implemented using “getAllMediaOptions”.

This library enables an easier development of controllers for automation systems, wherein a controller is a client of a group of NVTs. The industry partner is starting to make their controllers this way. Gradually they are integrating NVTs that explore the capabilities described in section II, to command alarms and other security systems based on video analysis techniques.

VII. CONCLUSION

This paper summarizes the knowledge that resulted from the implementation of an ONVIF library for NVT clients. Currently available toolkits were presented, and the selection of one discussed. We explain how to use the toolkit to develop a basic library, mainly consisting of ONVIF stubs and toolkit plugins. Then the implementation of several functionalities is discussed, demonstrating that much more value can be added, namely by integrating plugins and stubs, hiding details of both ONVIF and base WSs. This way the number of function parameters is minimized, greatly augmenting the simplicity of library API.

The proposed solutions address the fundamental challenges that we faced, and form patterns that are applicable to other ONVIF services, to provide higher level functionality in the entire library. This work has been validated in by an industry partner that is using a library applying all techniques here described. Future work includes adding support for multithreading and other ONVIF services besides NVT.

REFERENCES