MPL Binding Generation for Embedded Systems as Aspect Weaving

Mateus Krepsky Ludwich and Antônio Augusto Fröhlich

Laboratory for Software and Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil
{mateus,guto}@lisha.ufsc.br

Abstract

Programming Languages have a main role in computational systems development. Among them, the Managed Programming Languages (MPLs), from which JAVA and LUA are examples, provide developers with features to improve their productivity.

Several initiatives have been taken on the last decade in order to enable the use of MPLs not only for general propose systems as well for embedded systems fulfilling time and resource usage requirements impose by these systems. However, in order to be useful in embedded systems, MPLs must provide features for interacting with the environment where the embedded system is inserted on. Such interaction is usually implemented by using hardware devices, such as sensors and actuators, transmitters and receivers, and timers and alarms. The interface between hardware devices and MPL is commonly implemented by hand, which can lead to programming errors on the resultant binding code.

This paper presents a method to abstract hardware devices in order to be used by applications written using MPL in the embedded systems scenario. Hardware mediators are used to abstract and to organize hardware devices in a suitable manner for embedded systems fulfilling time and resource consumption requirements. By isolating hardware mediators from the specificities of an MPL, the problem of adapting a hardware device to work with a new MPL can be faced as an aspect weaving problem which can be automatically solved by using a proper tool.

The proposed method is evaluated on the MPLs JAVA and LUA in a case study encompassing H.264 video encoding. The obtained results corroborate the suitability of the proposed method on automatically adapt a hardware mediator to a new MPL while fulfilling requirements of performance and memory consumption.

Keywords: Binding Generation, Aspect-Oriented Programming, Embedded Systems, Foreign Function Interface

1. Introduction

A Managed Programming Language (MPL), from which JAVA and LUA are examples, is a kind of programming language which provide developers with features to improve their productivity. Productivity improvement is obtained by using constructions with a higher level of abstraction, enabling the developer to express and validate his ideas in a short period of time (such as object-orientation, domain specific constructions and APIs), and by features that make

the occurrence of programming errors less often, reducing the time spend on program debugging (such as automatic memory management, memory protection, and exceptions) [1, 2, 3, 4].

During the last decade several initiatives have been taken in order to enable the use of MPLs not only in general propose systems scenario as well in embedded systems scenario fulfilling the time and resource requirements impose by such systems. However, in order to be really useful for embedded systems MPLs must provide features for interacting with the environment where the embedded system is inserted on. Such interaction is usually implemented by using hardware devices. Sensors and actuators enable the system to interact with the environment. Transmitters and receivers are used for communicating with other systems. Timers and alarms are used to implement real-time operations.

The interaction between MPLs and hardware devices is performed by using the so called Foreign Function Interface (FFI). However, an FFI do not specify by itself how to abstract hardware or how to organize these abstractions. This work aims to fulfill this gap, introducing a method to interface hardware devices and applications written using MPLs in context of embedded systems. We propose a method to abstract such hardware devices and we show that the problem of adapting a hardware device to be used for an MPL can be faced as an aspect weaving problem, automatically generating the binding between the device and the language. Our method is automated by a tool which uses a meta-model to represent hardware device abstractions and FFI aspects.

The next sections of this paper are organized in the following way: Section 2 reviews how hardware devices can be abstracted and organized, how MPLs interact with hardware devices, and what are the approaches to facilitate such interaction. Section 3 makes a overview of the meta-model we have used in our tool. Section 4 introduces the proposed method for abstracting hardware devices and shows how the adaptation of a hardware device for a specific MPL can be solved as an aspect weaving. Section 5 presents our case study as well the obtained results on evaluating our proposal automatically adapting a hardware mediator to three FFIs (two for JAVA and one for LUA) while fulfilling requirements of performance and memory consumption. Finally, Section 6 presents our final considerations.

2. Related work

Two research fields are direct related to our work: how to abstract and organize hardware devices to be used by software, and how make these abstractions accessible to MPLs.

2.1. Hardware abstraction

Hardware Abstraction Layer (HAL) is an approach to abstract hardware specificities, providing for the operating system (OS) abstract hardware devices. The hardware devices abstraction is performed by services such as interruption handling, reinitialization handling, DMA transferences, timers control, and synchronization between multiprocessors [5].

In order to port a HAL implementation to a new hardware platform, it is necessary to implement all services provided by such HAL. However, many services provided by a HAL might not be used by an application and providing all these services generates unnecessary memory and performance overhead. An example of overhead caused by the way HAL are designed and implemented is the UCLINUX, an OS which target embedded systems and relies on a HAL. In such case, UCLINUX inherits from LINUX several of its features, among them, the file system abstraction. Such fact impacts not only on the OS size, as well in all system initialization infrastructure and application loading [6].

An alternative to monolithic HALs are the modularized or component-based HALs. An example of component-based HAL is find on the RedHat's eCos OS [7]. Although, the HAL used by eCos is based on software components it is not generated according to the application thus, it can carry unnecessary code for the system. In the case of SOCs generated from the micro-architecture LEON2, for example, the eCos system assumes the existence of an UART device what is not necessarily true [6]. As eCos, the AUTOSAR initiative proposes an approach based on a modularized HAL. However, AUTOSAR solves the problem of using HALs with unnecessary code by employing a generation process. During the configuration step, prior to linking, it is possible to select and combine basic software modules to create a application-oriented HAL [8].

The development of components for abstracting hardware devices can be guided by methodologies based on domain engineering. Such methodologies have been used for abstracting hardware devices without generating unnecessary interdependencies between the devices been abstracted. That is the case of the Application-Driven Embedded System Design (ADESD) methodology [9]. The Embedded Parallel Operating System (EPOS), represents the case study of the ADESD applied on the domain of operating systems. At EPOS the abstractions, obtained from domain decomposition, are OS abstractions such as threads, semaphores, abstractions for communication such as network, channels, and others. In order to provide system abstractions with the needed hardware support, ADESD defines the concept of *hardware mediator*. Hardware mediators sustain an interface contract between system abstractions and the machine, allowing for such abstractions to be machine-independent [6]. The Figure 1 shows hardware mediators, and their relation to system abstractions. Besides been componentized and application-driven, hardware mediators are designed to be efficient. By using metaprogramming techniques and function's inlining is possible to dissolve mediators among the abstractions that use it, which avoids time overhead in the use of mediators eliminating method call overhead.

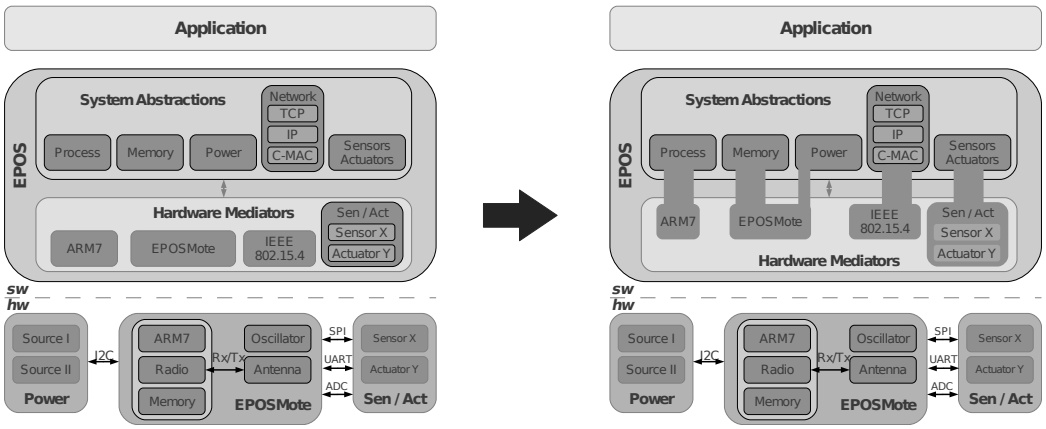

Figure 1: Hardware Mediators.

2.2. Hardware abstraction and MPLs

Automatic memory management is one of the main features provided by MPLs. Allocated objects are automatically freed after they are no longer needed, freeing developers from writing code to deallocate objects and eliminating the risk of memory leak. On the other hand, because

MPLs use automatic memory management, the object address is only know by the runtime support system (e.g. Virtual Machine) of the MPLs. Therefore, MPLs are unable to directly control memory mapped devices in the way languages such as C and C++ do (by using the concept of *pointer*). MPLs also do not provide the concept of *inline assembly*, used to control hardware devices by using dedicated I/O assembly instructions. In other to solve these limitations, MPLs use a mechanism called Foreign Function Interface (FFI).

FFI is a mechanism which allows for programs written in one language to use constructions of a program written in another. The language that defines the FFI is called *host language* and the language that has its constructions used is called *guest language*. The host language is usually a higher level language (i.e. MPL) and the guest language is usually a lower level language, such as C and C++. The FFI mechanism has a high importance to MPLs, not only new programs use FFI but also some of the main libraries of the MPL are implemented by using FFI. This is the case of the Java Standard Platform (JSE), which uses FFI to implement packages such as *java.io*, *java.net* and *java.awt*.

However, manual utilization of a FFI is error prone since the developer must take into consideration the semantics of two distinct languages (the host and the guest language). Besides that, many FFIs demand the developer to make the "*parsing*" of each native method manually. The following code illustrates the implementation of a sum method using the Java FFI KNI (a FFI used as based for many Java Virtual Machines - JVMs, targeting embedded systems) [10].

```
KNIEXPORT KNI_RETURNTYPE_VOID
Java_simplemath_Adder_sum() {
    jint a = KNI_GetParameterAsInt(1);
    jint b = KNI_GetParameterAsInt(2);
    KNI_ReturnInt(a + b);
}
```

In order to obtain the sum arguments, the developer should remember the order of the formal parameter declaration, represented by the indexes "1" and "2" passed to FFI functions.

Binding code generators solve the problem of manual parsing native methods arguments and deal with the semantic differences between the languages encompassed by an FFI, by generating the binding code (or part of it) from the native source code or from higher-level descriptions. The *SWIG* [11] tool and the Python's foreign function library *ctypeslib* [12], for example, generate binding code from C/C++ header files, avoiding the manual parsing of arguments. The former supports several languages as output such as Python, D, and Java, and the later focuses on Python programs. The *Jeannie* language encompass the semantics of C and Java therefore, it has the capacity of identify semantic errors of those two languages. A *Jeannie* program then, is used to automatically generate JNI wrappers with fewer semantic errors [13].

2.3. Discussion

Several FFI and binding code generators limitations had motivated us on the elaboration of a method to abstract hardware devices for MPLs. First, an FFI by itself does not guide the developers on the abstraction of hardware devices, it just provides for the developers a way to access constructions of other languages, such as *pointers* and *inline assembly*, capable of access hardware devices directly. Binding code generators make easy the use FFIs, solving the problem of manual parsing of native methods arguments and dealing with the semantic differences between host and guest language. However, as FFIs, binding code generators do not deal with the problem of device drivers abstraction.

The method presented in this paper uses the concept of hardware mediators to abstract hardware devices and a set of FFI focused on embedded systems to perform the interaction between hardware devices and MPLs. The binding code is generated from descriptions of hardware mediators and from descriptions of FFIs furthermore, the adaptation of a hardware mediator to a FFI is automatically performed, as an aspect *weaving* process.

A work close related to ours is presented by Schoeberl et al. [14]. By using the concept of *hardware object*, is is possible to access memory addresses from the real machine, allowing the implementation of device drivers in Java. This is performed by extending a special Java class (the *HardwareObject* class) which has special treatment from the point of view of the JVM. A modified version of the the JVM has special implementations for *getfield* and *putfield* bytecodes, capable of accessing hardware addresses outside the Java heap. However, the work of Schoeberl et al. focus only on Java, while our work focus on the reuse of the functional part of a binding code (i.e. the device driver) though several FFIs and MPLs, as explained in Section 4.

3. DERCS

Our method uses a intermediate representation to describe hardware mediators and FFI aspects. This section describes the meta-model we have used and Section 4 describes the method we have proposed.

The Model-Driven Engineering (MDE) proposes the development of complete computational systems from high level specifications which are transformed, though one of more phases, in order to generate the final system. In such context, the Distributed Embedded Real-Time Compact Specification (DERCS) is a specification/meta-model used to represent platform independent models. Such models, according to the Aspect-oriented Model-Driven Engineering for Real-Time systems (AMoDE-RT) methodology, are used together with the description of the target platform and mapping rules to generate the final system which encompass software and hardware elements [15].

According to the AMoDE-RT methodology, a DERCS model is generated from UML's class and sequence diagrams, and from diagrams that specify non-functional elements as aspects. Therefore, the DERCS meta-model defines structural elements, behavioral elements, and aspect-oriented elements, encompassing distinct visions into a single model.

Figures 2 and 3 show, respectively, the structural and behavioral elements defined by DERCS. Among the structural elements are classes, attributes, methods, and parameters. The behavioral elements determine the behavior of a method, detailing which messages such method can send to objects, which are its local variables, actions, etc. The structural and behavioral elements of DERCS have semantics similar to the semantics of object-oriented programming languages.

Figure 4 shows the aspect-oriented elements defined by DERCS. Among the aspect-oriented elements are aspects, pointcuts, structural adaptations, and behavioral adaptations. Those elements correspond to the concepts of Aspect-Oriented Programing (AOP), as defined by [16] whereas, aspect adaptations (*AspectAdaptation*) represent the concept of *advices*. An aspect adaptation can be structural (*StructuralAdaptation*) or behavioral (*BehavioralAdaptation*). Structural adaptations modify the structure of the elements of the model, for example, adding methods to a class or parameters to a method. On the other hand, behavioral adaptations modify the behavior of the elements of the model, for example, executing tasks before or after the execution of a method behavior, or completely changing a method's behavior.

The aspect-oriented elements of DERCS meta-model can be used to modify the interface of a component in order to adapt such component according to what is required by its client. As next

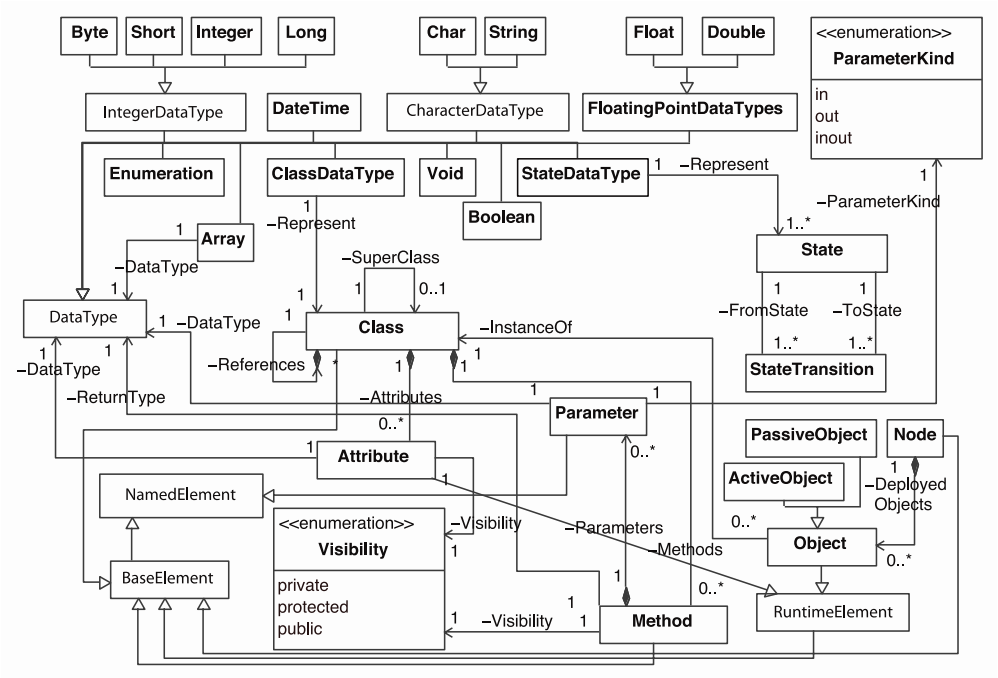

Figure 2: DERCS structural elements. Adapted from [15].

section shows, we have explored this feature of DERCS in order to adapt a hardware mediator interface for what is expected by a specify FFI.

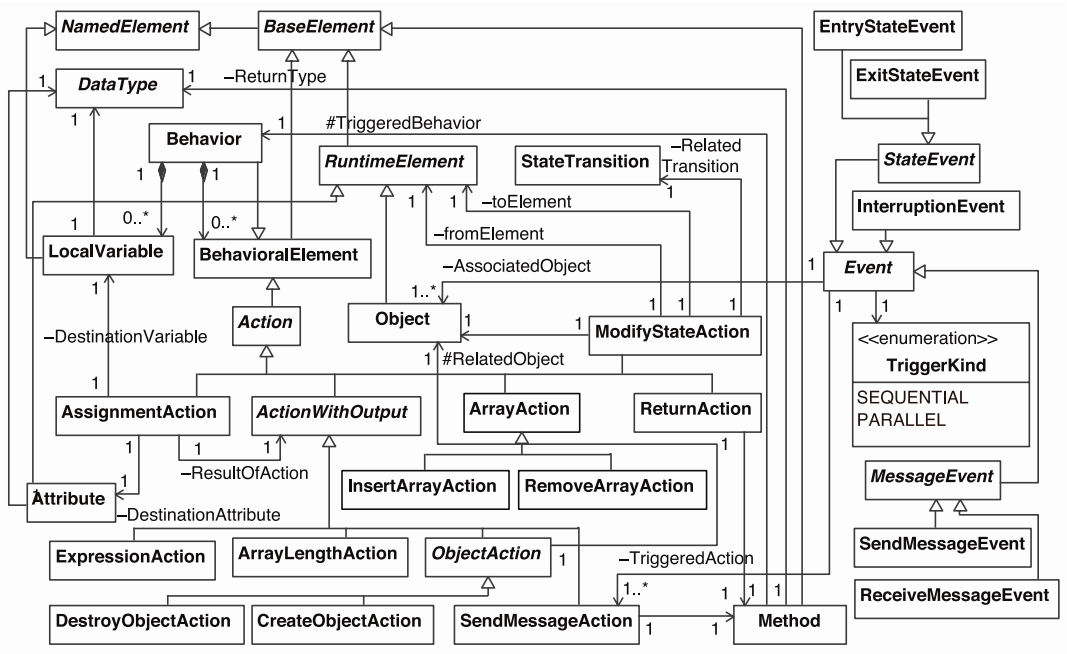

Figure 3: DERCS behavioral elements. Adapted from [15].

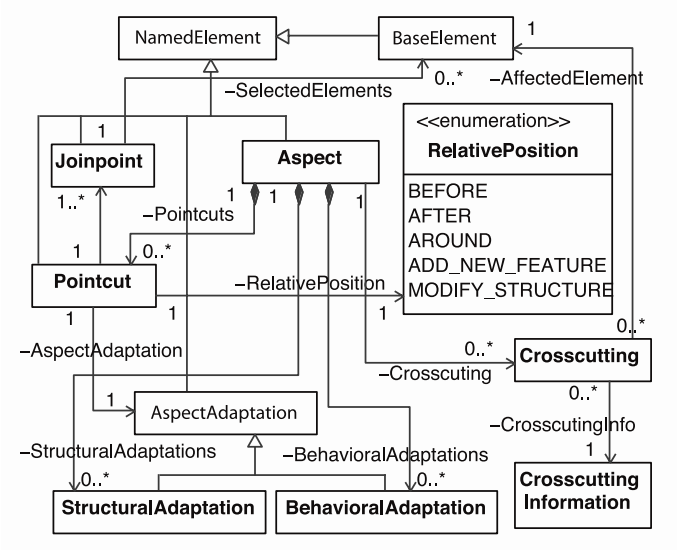

Figure 4: DERCS aspect-oriented elements. Adapted from [15].

4. Proposal

This section describes our method to abstract hardware devices to be used by programs written in distinct MPLs, adapting a hardware device to the target MPL as an aspect weaving. This section also shows the application of the proposed method for the languages JAVA and LUA.

We have applied our method, in the case of the JAVA language, to two JVMs that target embedded systems: KESO and NanoVM. KESO is a JVM targeting embedded control units of automotive systems [17]. NanoVM is a JVM targeting embedded systems based on the AVR8 architecture [18]. In the case of LUA, we have applied our method on the standard LUA implementation [19]. KESO is based on a generational approach, translating all JAVA bytecode to C and then to native code before the program execution, and also generating all the JVM runtime needed by the application that is going to be used. On the other hand, NanoVM performs bytecode interpretation, although it uses a special version of bytecode, which is optimized before the program execution. The standard LUA implementation can either compile the source code while it is running or previously and then, execute the compiled bytecode while the program is executing.

4.1. Necessity of automation

As shown in Section 2, automate the generation process of a binding code is desirable since it avoids programming errors on manually using an FFI such as the parameter types and order of the methods and due to the semantical differences between the two languages evolved in the FFI. For this reason the proposed method for abstracting hardware devices was automated by a tool called *Extensible Binding Generator* (EBG).

The Figure 5 presents the generation flow of a system according to our proposal for abstracting hardware devices for MPLs in the embedded system scenario. From the application analysis is possible to identify the hardware mediators that the embedded application is going to use. Then, the needed hardware mediators are selected and used as input to EBG. The other EBG's input is the selection of the target MPL and FFI. EBG has as outputs binding code already tailored for the target FFI, and the MPL counterpart of the hardware mediators.

The MPL counterpart of the hardware mediators is compiled together with the application by a standard MPL compiler, generating the correspondent bytecode. This bytecode can be externalized as is the case of the *class* files of JAVA or direct send to the MPL interpreter, as is the default case for LUA. The binding code generated by the EBG and the application bytecode are then integrated to the runtime support system (usually a virtual machine) of the target MPL. Finally, the runtime support system of the MPL which includes EPOS and the hardware mediators are compiled by GCC, generating the final system image.

4.2. Achieving reuse between distinct FFIs

The functional part of a binding code, which controls the hardware device, is independent of the target FFI. Therefore, it should be possible to reuse the functional part of a binding for distinct FFIs. In our approach, the reuse of a binding code among distinct FFIs is obtained by factoring it in functional part (FFI independent) and in non-functional part (FFI dependent), and by composing these parts in order to generate the final binding code which is going to be used by the target FFI. The functional part of a binding code, responsible for hardware device control, corresponds to a hardware mediator and its methods. The non-functional part of a binding code, which is dependent of the target FFI, corresponds to the methods and concepts defined by the API of the target FFI. In our approach, the non-functional part is represented as aspects of the

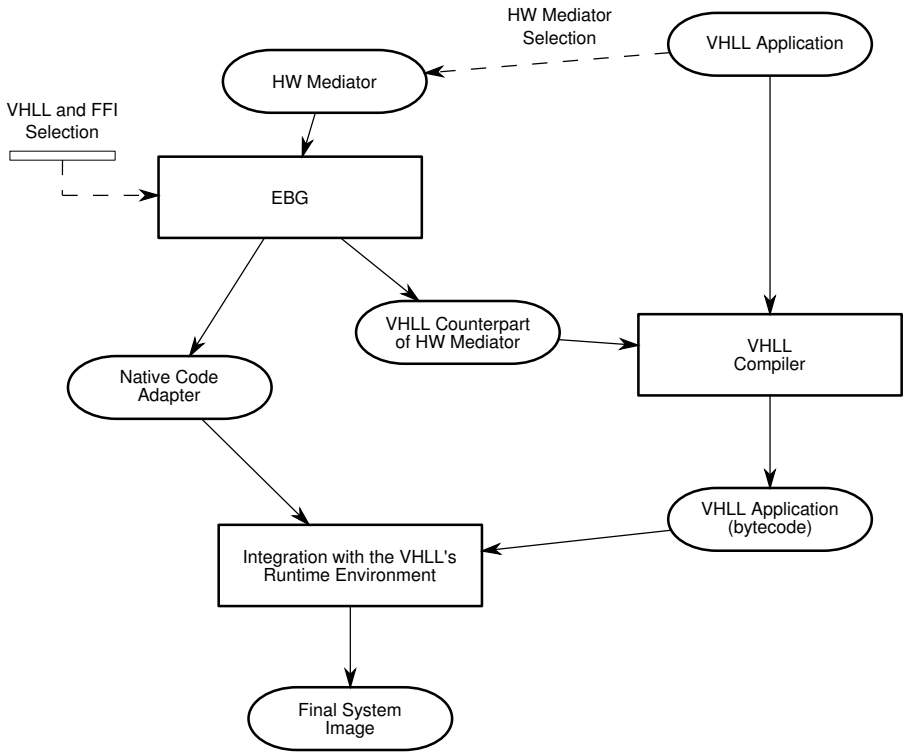

Figure 5: Binding code generation flow.

AOP, and are used to describe FFIs API and rules. The functional part (hardware mediator) is the component adapted by the FFIs aspects, by a AOP weaving process, generating the binding code for the target FFI.

Both hardware mediators and FFI descriptions are represented in our approach by an intermediary representation based on the DERCS meta-model, described in Section 2. The main advantage of using a the DERCS meta-model to describe hardware elements and FFIs, rather than perform just syntactic transformations directly on source code, is that DERCS provide us with data types with defined semantics, avoiding generating binding code with semantical errors.

The structural and behavioral elements of DERCS are used to describe the hardware mediators. The aspect-oriented related elements of DERCS are used to describe the FFI aspects. Although, it is worth mentioning that not all DERCS elements where used for EBG internal representation because the primary focus of EBG is not MDE therefore, elements such as *RuntimeElement* were left out. Still, it is possible to integrate EBG in a MDE tool such as the *GenERTiCA* tool [15], where the DERCS meta-model was primary used.

Another difference between the EBG and the original use of DERCS in *GenERTiCA*, is related to the composition of aspects with structural and behavioral elements. As *GenERTiCA* focus on MDE, the aspect weaving occurs during the phase of system code generation. In such case, the DERCS meta-model specifies in an abstract form which structural and behavioral adaptations should occur, however, the semantics of such adaptations are described by the rules responsible for the mapping from model to code. On the oder hand, in EBG, the aspect weaving

occurs at the model level therefore, we have extended DERCS to support concrete structural and behavioral adaptations with semantics defined at the model level.

Figure 6 shows how we have extended the structural adaptation *StructuralAdaptation* of DERCS. We have defined two new concrete types of structural adaptation, *DeclareParent* and *AddAttribute*. The semantics of *DeclareParent* is to define a new superclass for the class affected by the aspect. In such way, the target class of the adaptation (the one affected by the aspect) has its superclass changed to the class indicated by the attribute *parent* of *DeclareParent*. The adaptation *AddAttribute*, when applied, adds a new attribute to the target class of the adaptation.

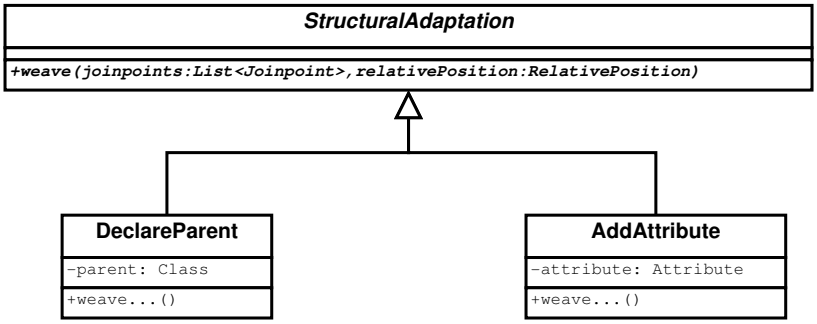

Figure 6: Extensions of DERCS's structural adaptation.

Figure 7 shows how we have extended the behavioral adaptation *BehavioralAdaptation* of DERCS. We have defined two new concrete types of behavioral adaptation, *AddSendMessageAction* and *AddReturnSendMessageAction*. The adaptation *AddSendMessageAction* adds behavior into a method's body, more specifically, it adds a call action (represented by the behavioral element *SendMessageAction*) to a method bellowing to a certain class. *AddSendMessageAction* specifies the message to be added, the related method, and the involved classes. The adaptation *AddReturnSendMessageAction* is identical to *AddSendMessageAction* except that, besides adding a message of call to a method, it considers that such message is going to return a value, and this value will be returned by the method affected by the aspect. Both adaptations *AddSendMessageAction* and *AddReturnSendMessageAction* can be applied at the beginning of a method's behavior (*BEFORE*), at the end (*AFTER*), or they can completely replace the method's behavior (*AROUND*).

The composition of a hardware mediator and FFI aspects, both represented using our adapted version of DERCS, is performed by the *Weaver* module of EBG. Figure 8 shows the internal organization of EBG. For each mediator selected as an EBG input are applied the aspects of the selected FFI, adapting the mediator to the FFI's API. The output of the *Weaver* module then enters in the *Generator* module which transforms the internal EBG representation in source code written in the programming language required by the target FFI.

The weaving process, performed by the *Weaver* module of EBG is described by the Algorithm 1.

For each element of the model that should be affected (i.e. for each *joinpoint*), which is described by the aspect's *pointcuts*, are applied all correspondent aspect adaptations. As it is modeled, each *pointcut* knows all elements of the model that are going to be affected, as it knows all aspect adaptations that should be applied. In such way, each aspect adaptation application over a model element is performed by the invocation of the *weave* method of the adaptation.

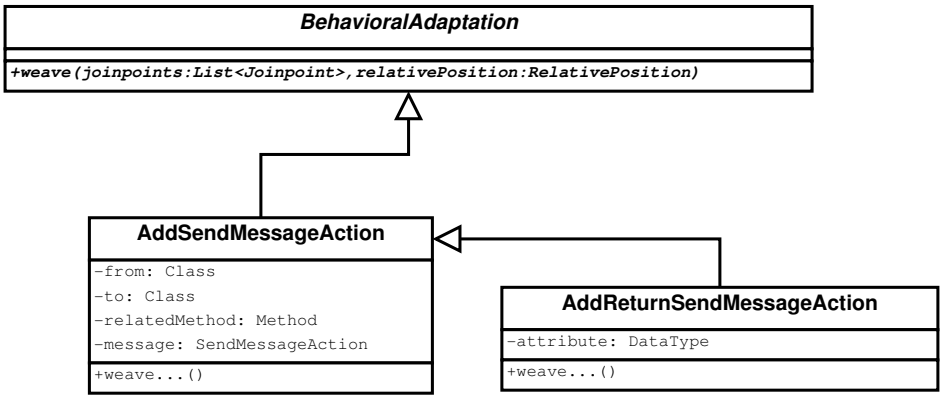

Figure 7: Extensions of DERCS's behavioral adaptation.

Algorithm 1 Aspect weaving in EBG.

```
for all aspects ∈ model do
  aspect ← model.aspect
  for all pointcuts ∈ aspect do
    adaptation ← pointcut.aspectAdaptation
    joinpoints ← pointcut.joinpoints
    relativePosition ← pointcut.relativePosition
    adaptation.weave(joinpoints, relativePosition)
  end for
end for
```

During the elaboration of this work, we have identified two patterns used to adapt a hardware mediator to the target FFI. These patters were named *Object-Based Adaptation* (OBA), and *Class-Based Adaptation* (CBA). OBA and CBA define how the relation between the binding code and the hardware mediator is going to be. For the OBA pattern, the binding code class is going to aggregate the hardware mediator class thus, a binding object is going to have a reference to a hardware mediator object. For the CBA pattern, the composition between binding code and hardware mediator is performed by inheritance thus, a binding code class is going to extend a hardware mediator class.

Figures 9 and 10 illustrate in an abstract form (independent of the target FFI) the weaving process using, respectively, OBA and CBA patterns. Before executing any weaving, EBG performs a set of preparations using the hardware mediator description (*HW_Mediator* class). First, from *HW_Mediator* EBG generates the MPL counterpart of the hardware mediator (represented by the *MPL_Mediator* class) which is going to have all methods of the hardware mediator but such methods will not have implementation, they will be all marked as *native*. Also, from the analysis of *HW_Mediator*, EBG generates a skeleton for the binding code class (represented by the *NativeMediator* class) which has the same methods as *HW_Mediator* but also without implementation.

While weaving following the OBA pattern, after the weaving process, the binding code class *NativeMediator* is going to have a reference to the hardware mediator *HW_Mediator*. Such result is obtained applying the structural adaptation *AddAttribute* on *NativeMediator*. The reference to

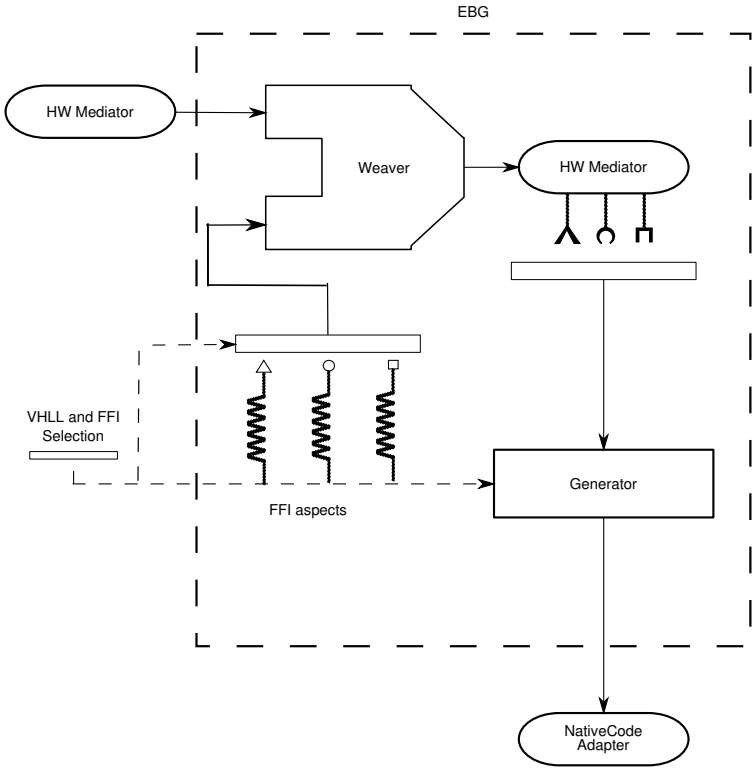

Figure 8: EBG architecture.

HW_Mediator is then used to implement the *NativeMediator*'s methods using *AddSendMessage-Action* adaptations, according to the target FFI specifications.

Figure 11 shows the application of the OBA pattern using the FFI aspects related to the NanoVM's FFI. Figure 12 shows the same for the KESO's FFI. Both figures show the equivalent to the rightmost side of the Figure 9, i.e., the result of the weaving process. Looking on the implementation of the *operation* method, disregarding the differences (which are FFI dependent), it is possible to notice that in both cases the *HW_Mediator operation* method is invoked though the *inner* attribute, which is a reference to the *HW_Mediator* object.

While weaving following the CBA pattern, after the weaving process, the binding code class *NativeMediator* is going to extend the hardware mediator *HW_Mediator* class. Such result is obtained applying the structural adaptation *DeclareParent* on *NativeMediator*. Then, the *Native-Mediator* methods are implemented using *AddSendMessageAction* adaptations, according to the target FFI. In the case of CBA, the code snippet *inner.operation* of Figures 11 and 12 is replaced by *HW_Mediator::operation*, which represents the method inherited from *HW_Mediator*.

The output of EBG's *Weaver* module is a set of binding code classes already adapted to the target FFI, however those classes are still described using the internal representation of EBG (the modified version of DERCS). The *Generator* module of EBG then, is the responsible to transform those binding code classes into source code that can be integrated to the runtime environment of the target MPL. The transformation process performed by the *Generator* module

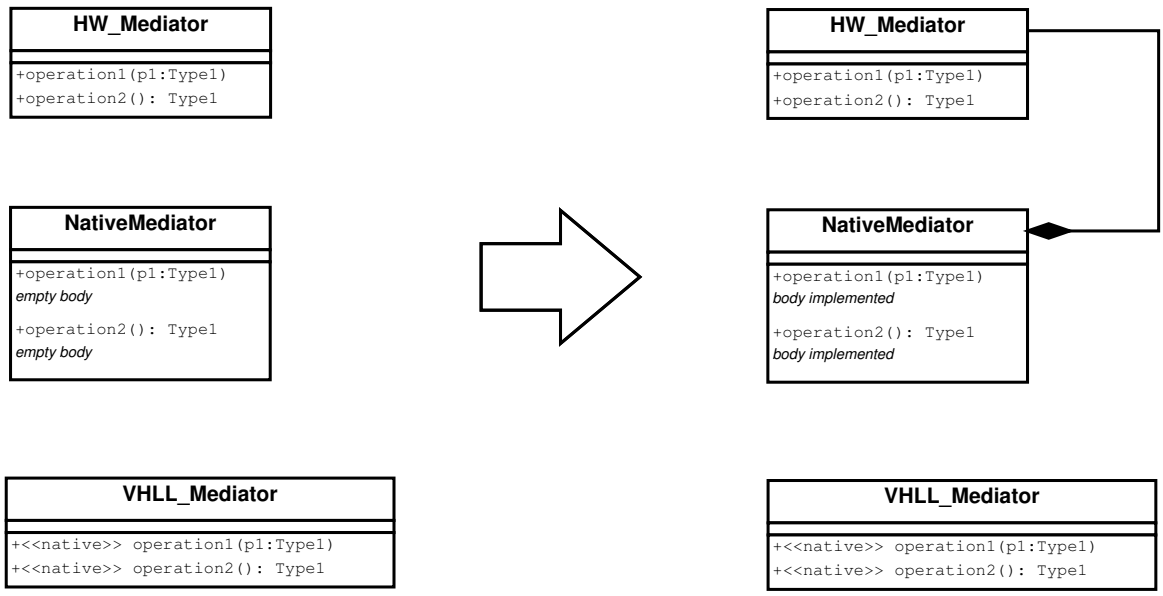

Figure 9: Application of Object-Based Adaptation.

is shown in Algorithm 2 and consists on the translation of methods's bodies to the language expected by the target FFI, and on the applications of one or more templates to format the generated code according to the rules of the target FFI.

Algorithm 2 Binding code generation on EBG.

for all $method \in NativeMediator$ **do**
  $methodBody \leftarrow toTargetLanguage(method.body)$
  $bindingCode \leftarrow templateFFI(methodBody)$
  $write(bindingCode)$
end for

In the case of KESO, the output of the *Generator* module of EBG is composed by the JAVA files containing, respectively, the *weavelet* class and the JAVA counterpart class of the hardware mediator. The *weavelet* class is then used by the KESO compiler to generate the final system. The JAVA counterpart class of the hardware mediator is used by the application, and it is compiled by a standard JAVA compiler. The resultant bytecode is then translated to C by the KESO compiler during the system generation.

As an example of code generate by EBG to KESO's FFI, Figure 13 shows the main parts of the weavelet class. It can be notice that the code snippet

```
return operation(inner, a, b);
```

was generated from the *operation* method's body of the, shown in Figure 12. In such case, the EBG internal representation was translated to code written in C language, specifying exactly the source code that is going to be generated by the KESO compiler. As KESO translate JAVA bytecode to C and in this specific case the generated code is also in C rather then in C++, we have

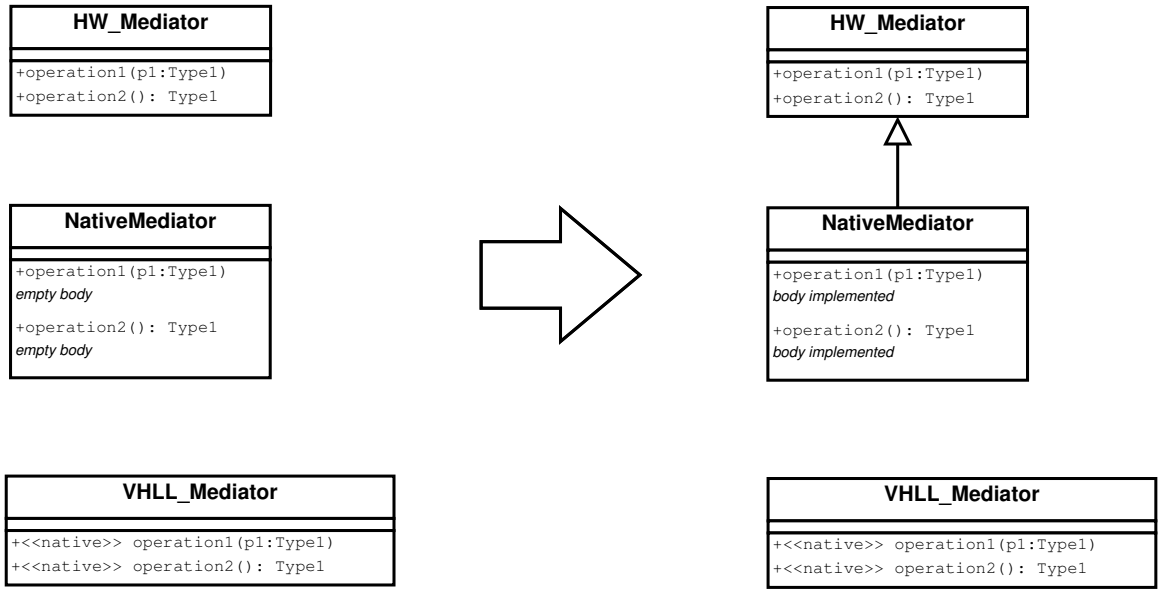

Figure 10: Application of Class-Based Adaptation.

used a C to C++ wrapper in order to access the methods of EPOS's hardware mediator which are written in C++. This wrapper "translate"

```
return operation(inner, a, b);
```

in to

```
return inner->operation(a, b);
```

It is possible to notice in Figura 13 the use of the OBA pattern, as the pointer *inner* is one of the attributes of the generated binding code class.

In the case of NanoVM and the Lua VM, which are based on bytecode interpretation and do not use a generation process as KESO, the output of the *Generator* module of EBG is already the C++ code implementing the binding code that is statically linked (compiled together) the respective VMs. Also for those cases, the *Generator* module generates the respectively Java and Lua files that represent the MPL counterpart of the mediator and can be imported by the MPL application. As NanoVM and Lua VM we have used were already ported to C++, there was no need of using C/C++ wrappers to interact with the mediators classes of EPOS.

4.3. Achieving performance and efficient resource usage

Ideally, the response time of a hardware device accessed by using a binding code should be the same than the response time obtained when accessing the same device using languages such as C and C++ which support direct hardware access. The usage of resources such as memory due to the use of a binding code, similarly to response time, should be kept at the minimum to not impact in the overall resource usage of the system.

Performance and efficient resource usage is achieved by using the concept of *hardware mediators*, presented at Section 2. Hardware mediators allow for development of efficient binding

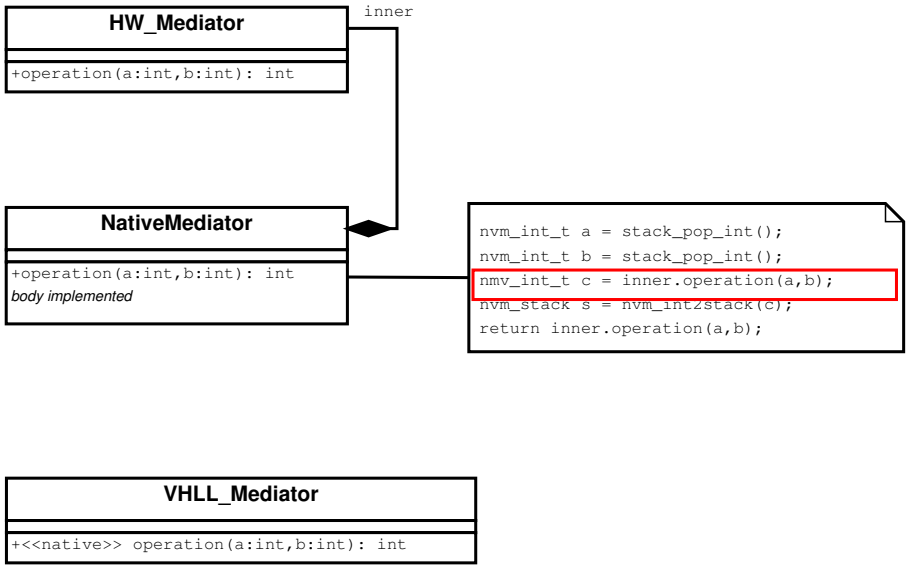

Figure 11: Application of Object-Based Adaptation for NanoVM's FFI.

code due two reasons. The first reason is related to the design of hardware mediators, since there is a mediator for each device in the platform, it is eliminated the overhead caused by monolithic HALs. The second reason is related to the techniques used to implement the hardware mediators. Using static metaprogramming techniques is possible to dissolve mediators among the abstractions that use it, eliminating method call overhead for the mediator methods.

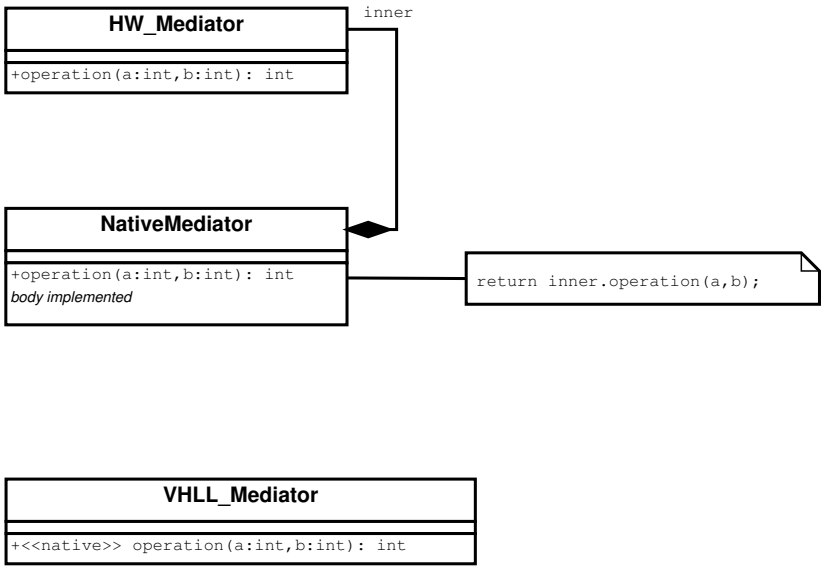

Figure 12: Application of Object-Based Adaptation for KESO's FFI.

```
public class MediatorWeavelet extends Weavelet {
  //  ...
  public boolean affectMethod(IMClass clazz,
    IMMethod method,
    Coder coder) throws CompileException
  {
    if (method.termed("operation01(II)I")) {
      coder.addln("return operation(inner, a, b);");
      return true;
    }
    //  ...
    return false;
  }
}
```

Figure 13: *Weavelet* class of a generic hardware mediator.

5. Case study

This section presents the evaluation of the proposal of binding generation to interface hardware devices and embedded MPL, introduced at Section 4. In order to evaluated our proposal we have implemented it in a case study encompasing distributed motion estimation for digital video encoding. We have evaluate our proposal according to the reuse of binding code classes among distinct MPLs and FFIs, showing that the functional part of a binding code can be reused at distinct MPLs and FFIs without modifications. In order to demonstrate the feasibility of the generated binding code we also we have evaluated they according to performance, and memory consumption.

5.1. Distributed Motion Estimation

We have evaluated the application of our approach for binding generation over a component to perform *Motion Estimation* (ME) for H.264 video encoding. ME is a technique employed to explore the similarity between neighboring pictures in a video sequence. Figure 14 illustrates the ME process for the neighboring pictures *A* and *B*. By searching for similarities between these two pictures, it is possible to determine which blocks from picture *A* are also found in picture *B*. Such displacement of picture blocks is encoded by *motion vectors* (represented by the small arrows in the bottom side of Figure 14). Exploring the similarity between neighboring pictures allows for difference-based encoding, thus increasing the compression rate of the generated bitstream [20]. As ME is a significant stage for H.264 encoding, since it consumes around 90% of the total time of the encoding process [21], the component we have adapted to work with embedded JAVA and LUA uses an optimized version of ME. In order to improve the performance of ME, the component uses a data partitioning strategy where the motion estimation for each partition of the picture is performed in parallel in a specific functional unit, such as a core of a multicore processor. However such complexity is hidden from the point of view of the user of the component (e.g. H.264 encoder), which only sees a component to perform ME though a *match* method.

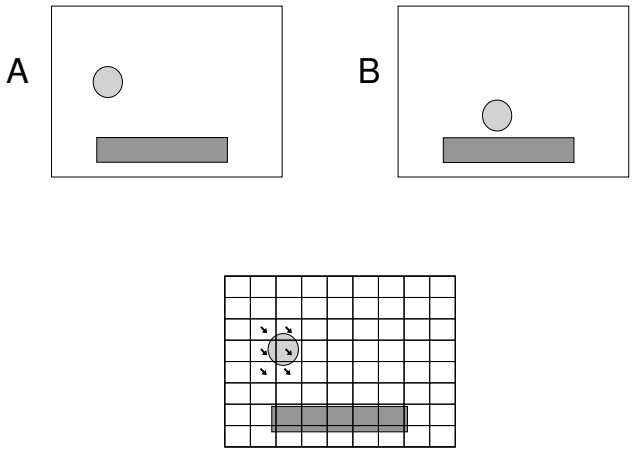

Figure 14: Motion Estimation

Such component for ME computation was developed for the project the Brazilian project Rede H.264, which aims to develop standards and products for the Brazilian Digital Television

```
public class DmecApp extends Task {
  public void launch() {
    int width = 1920;
    int height = 1088;
    PictureMotionEstimator pme;
    pme = new PictureMotionEstimator(width, height);

    Picture cp, rp;
    // cp and rp pictures selection ...

    PictureMotionCounterpart mvsc;
    mvsc = pme.match(cp, rp);

    // Results processing...
  }
}
```

Figure 15: Motion Estimation Java application.

[22]. One of the goals of the project Rede H.264 is the integration between its components. That is the case of the set top boxes, where interactive user applications written in Java using the Ginga-J [23] middleware are integrated with encoders and decoders written in C, C++, and in hardware description languages.

In order to use the ME component from JAVA and from LUA, first we wrote an application that mimics the behavior of an H.264 encoder then, we have generated binding code classes for JAVA and for LUA.

Figure 15 shows the Java version of the encoder-like application, Figure 16 shows the same application written in LUA. The application provides the component with pictures, get from the component the ME results (motion vectors and motion cost), and checks if the results are correct.

Using *EBG* we have generated binding code classes for the ME component in order to integrate it with JAVA and LUA. Figure 17 shows the binding code wrapping the *match* method to provide it for LUA. It recovers the motion estimator object from the LUA's stack, as well the ME parameters: the *current* and the *reference* picture (which correspond, respectively, to picture *A* and *B* from Figure 14). Then, it invokes the *match* method which returns the motion vectors and the motion cost (grouped together by the class *PictureMotionCounterpart*). Finally, it puts back in the LUA's stack the ME's result and returns the control to the LUA runtime. Figure 18 shows the same binding code for the NanoVM FFI. Its structure is very similar to the LUA binding code: it invokes the motion estimator and it uses the NanoVM stack to get the parameters and publish the ME return. The output generated by EBG for KESO FFI, shown at Figure 19, is different from the output for LUA and NanoVM FFI due to the generational approach of KESO. Instead of generating the binding code, EBG generates a *weavelet* class (following the KESO FFI) which is then used by the KESO compiler to generate the final final binding code. It is important to point out that, for all the binding code classes, the functional component of the binding is the same for all the FFIs, therefore demonstrating its reuse.

In order to evaluate the feasibility of the generated binding code classes we have evaluated them according to performance and memory consumption. For performance evaluation, first we have measured the time overhead caused by the binding code of the *match* method. Then, we have calculated how such overhead affects the ME throughput in order to see the impact in ME

```
function launch()
  w = 1920
  h = 1088

  pme =
   PictureMotionEstimator:new{width = w, height = h}

  -- cp and rp pictures selection ...
  mvsc = pme:match(cp, rp)

  -- Results processing...

end

launch()
```

Figure 16: Motion Estimation Lua application.

```
int pme_native_match(lua_State* L)
{
  PictureMotionEstimator* pme =
     static_cast <PictureMotionEstimator*>(lua_touserdata(L, 1));

  MEC_Picture* currentPicture =
     static_cast <MEC_Picture*>(lua_touserdata(L, 2));

  MEC_Picture* referencePicture =
     static_cast <MEC_Picture*>(lua_touserdata(L, 3));

  PictureMotionCounterpart* pmc;
  pmc = pme->match(currentPicture, referencePicture);

  lua_pushlightuserdata(L,  static_cast <void*>(pmc));

  return 1;
}
```

Figure 17: Binding code for *match* method (Lua FFI)

```
void native_picture_motion_estimator_invoke(u08_t mref)
{
  // ...
  else if (mref == NATIVE_METHOD_PME_MATCH) {
    NativePicture* referencePicture =
      NativePictureMap::getInstance()->get(stack_pop());
    NativePicture* currentPicture =
      NativePictureMap::getInstance()->get(stack_pop());
    NativePictureMotionEstimator* npme =
      NativePictureMotionEstimatorMap::getInstance()->get(stack_pop());

    npme->match(currentPicture, referencePicture);
  }
  else {
      error(ERROR_NATIVE_UNKNOWN_METHOD);
  }
}

NativePictureMotionCounterpart* match(NativePicture* currentPicture,
NativePicture* referencePicture)
{
  __nativePMC->set_inner(
    __inner->match(currentPicture->inner(), referencePicture->inner()));
  return __nativePMC;
}
```

Figure 18: Binding code for *match* method (NanoVM FFI)

```
public class OBA_C_PME_Weavelet extends Weavelet {
  //  ...
  public boolean affectMethod(IMClass clazz,
    IMMethod method, Coder coder) throws CompileException
  {
    if (method.termed(
"match(Ldmec/Picture;Ldmec/Picture;)Ldmec/PictureMotionCounterpart;"))
    {
      IMMethodFrame frame = method.getMethodFrame();
      IMSlot[] arguments = frame.getMethodArguments();
      assert(arguments.length == 3);
      IMSlot thiz = arguments[0];
      IMSlot currentPicture = arguments[1];
      IMSlot referencePicture = arguments[2];

      String _thiz =
          WeaveletUtility.mountArg("PictureMotionEstimator*", thiz);
      String _currentPicture =
          WeaveletUtility.mountArg("MEC_Picture*", currentPicture);
      String _referencePicture =
          WeaveletUtility.mountArg("MEC_Picture*", referencePicture);

      String _args = _thiz + ", " + ", " + _currentPicture + ", " +
          _referencePicture;
      coder.addln(_thiz + "pme_match(" + _args + ");");
      coder.addln("return " + thiz + "->__pmc;");

      return true;
    }

    //  ...

    return false;
  }
}
```

Figure 19: Binding code for *match* method (KESO FFI)

Table 1: Time overhead

FFI	Device (μs)	Binding Overhead (μs)	VM Overhead (μs)
Lua	9.68362E+005	2.73753E-003	6.29973E+001
NanoVM	9.68362E+005	3.97423E-003	1.96960E+001
KESO	9.68362E+005	8.26280E-003	1.18113E+003

Table 2: ME throughput

FFI	Original Throughput (FPS)	New Throughput (FPS)
Lua	1.032672	1.032604
NanoVM	1.032672	1.032651
KESO	1.032672	1.031416

overall performance. For memory usage evaluation, first we have measured how many bytes the binding code for the *match* method has. Then, we have calculated how much of the total system size is caused by such overhead.

Table 1 shows the time overhead of the *match* method for each FFI used. The column *Device* contains the device time, which is the time of the *match* method while accessed directly through a C++ program, without the use of any binding. Such time is independent of the FFI used. The column *Binding Overhead* shows the overhead just for the binding code of the match method, and column*VM Overhead* shows the runtime enviroment support (e.g. virtual machine) overhead for decoding an instruction of native method invoking.

In order to evaluate the time overhead impact on the overall ME performance we have calculated how such overhead affects the ME throughput. Table 2 shows the results. The column *Original Throughput* shows the original ME throughput (without using any binding code), while the column *New Throughput* shows the ME throughput while using each FFI/MPL evaluated. Considering that the required throughput for the ME is one Frame-per-Second (FPS), the new throughput is acceptable for all FFIs considered. The overhead caused by the binding code and the VM are independent of the *Device* time (they only depend on the number and the type of methods arguments and method return plus the VM decoding time of a native method invoke instruction). Thus, even considerating a faster ME implementation (e.g 30 FPS), all FFIs used will continue to meet the time requirements.

In order to evaluate the memory overhead caused by the generated binding code classes, we have measured how many bytes the binding code for the *match* method has. Table 3 shows the obtained values (column *Overhead*) for each FFI used. The same table shows the total system footprint, including application, and the MPL runtime (composed by the VM and EPOS), and how much of this size is caused by such the binding code classes. The memory available for the platform where DMEC runs is far away bigger than the new footprint obtained while using the binding code classes. Thus, for all FFIs, the generated binding code classes fulfill the memory usage requirements.

Table 3: Memory overhead and its impact on the total footprint

FFI	Overhead (byte)	Footprint (byte)	Impact (%)
Lua	800	1576512	0.05
NanoVM	1453	1459501	0.1
KESO	201	1463201	0.01

6. Conclusions

This paper introduces a method used to abstract hardware devices in order to be used by MPLs applications in the embedded system scenario. This method can be used for constructing hardware components libraries for embedded MPLs applications. It is also presented EBG, a tool which automates the task of generating native code adapters. The EBG shows that, when hardware devices and FFIs are properly factorized, the adaptation of a hardware device to a MPL can be treated as an aspect weaving problem thus, enabling the reuse of native code adapters by distinct FFIs.

The proposal was applied for embedded JAVA and embedded LUA, in a case study encompasing distributed motion estimation for digital video encoding. In the case of JAVA we have applied our proposal on two kinds of JVM and FFIs: the FFI of KESO VM which uses a generation approach and ahead-of-time compilation strategy translating the Java bytecode to C and the FFI of NanoVM which is based on bytecode intrepretation. In the case of LUA, our proposal was evaluated for the standard LUA FFI.

The mesuread overhead of time and memory shows that the binding code generated by using the proposed approach is small and keeps the requirements, which indicates that our approach if feasible for embedded systems.

[1] M. D. Bond, K. S. McKinley, Leak pruning, in: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems, ASPLOS '09, ACM, New York, NY, USA, 2009, pp. 277–288. doi:http://doi.acm.org/10.1145/1508244.1508277.
URL http://doi.acm.org/10.1145/1508244.1508277

[2] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, J. Vitek, High-level programming of embedded hard real-time devices, in: Proceedings of the 5th European conference on Computer systems, EuroSys '10, ACM, New York, NY, USA, 2010, pp. 69–82. doi:http://doi.acm.org/10.1145/1755913.1755922.
URL http://doi.acm.org/10.1145/1755913.1755922

[3] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, K. S. McKinley, Looking back on the language and hardware revolutions: measured power, performance, and scaling, in: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, ASPLOS '11, ACM, New York, NY, USA, 2011, pp. 319–332. doi:10.1145/1950365.1950402.
URL http://doi.acm.org/10.1145/1950365.1950402

[4] G. Phipps, Comparing observed bug and productivity rates for java and c++, Softw. Pract. Exper. 29 (4) (1999) 345–358. doi:10.1002/(SICI)1097-024X(19990410)29:4¡345::AID-SPE238¿3.0.CO;2-C.
URL http://dx.doi.org/10.1002/(SICI)1097-024X(19990410)29:4<345::AID-SPE238>3.0.CO;2-C

[5] A. S. Tanenbaum, Modern Operating Systems, 3rd Edition, Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[6] F. V. Polpeta, A. A. Fröhlich, Hardware mediators: a portability artifact for component-based systems, in: In Proceedings of the International Conference on Embedded and Ubiquitous Computing, volume 3207 of LNCS, Aizu,Japan, Springer, 2004, pp. 271–280.

[7] A. Massa, Embedded Software Development with eCos, 1st Edition, Prentice Hall PTR, 2002.

[8] A. development partnership, Autosar (automotive open system architecture), [Online; accessed May 28, 2012].

[9] A. A. Fröhlich, Application-Oriented Operating Systems, no. 17 in GMD Research Series, GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[10] I. Sun Microsystems, K Native Interface (KNI), Sun Microsystems, Inc., 2002.

[11] Simplified wrapper and interface generator.
URL http://www.swig.org/

[12] ctypeslib - useful additions to the ctypes ffi library.
URL http://pypi.python.org/pypi/ctypeslib/

[13] M. Hirzel, R. Grimm, Jeannie: granting java native interface developers their wishes, in: OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, ACM, New York, NY, USA, 2007, pp. 19–38. doi:http://doi.acm.org/10.1145/1297027.1297030.

[14] M. Schoeberl, S. Korsholm, T. Kalibera, A. P. Ravn, A hardware abstraction layer in java, ACM Trans. Embed. Comput. Syst. 10 (4) (2011) 42:1–42:40. doi:10.1145/2043662.2043666.
URL http://doi.acm.org/10.1145/2043662.2043666

[15] M. A. Wehrmeister, An Aspect-Oriented Model-Driven engineering approach for distributed embedded Real-Time systems, Ph.D. thesis, Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação., Porto Alegre, RS, Brazil (2009).

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, J. Irwin, Aspect-oriented programming, in: ECOOP, SpringerVerlag, 1997.

[17] I. Thomm, M. Stilkerich, C. Wawersich, W. Schröder-Preikschat, Keso: an open-source multi-jvm for deeply embedded systems, in: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '10, ACM, New York, NY, USA, 2010, pp. 109–119. doi:http://doi.acm.org/10.1145/1850771.1850788.
URL http://doi.acm.org/10.1145/1850771.1850788

[18] T. Harbaum, The NanoVM - java for the AVR (2005).
URL http://www.harbaum.org/till/nanovm/index.shtml

[19] Lua.org, Lua: about (2011).
URL http://www.lua.org/about.html

[20] T. Wiegand, G. J. Sullivan, G. Bjontegaard, A. Luthra, Overview of the h.264/avc video coding standard, Circuits and Systems for Video Technology, IEEE Transactions on 13 (7) (2003) 560–576. doi:10.1109/TCSVT.2003.815165.
URL http://dx.doi.org/10.1109/TCSVT.2003.815165

[21] X. Li, E. Li, Y.-K. Chen, Fast multi-frame motion estimation algorithm with adaptive search strategies in h.264, Vol. 3, 2004, pp. iii – 369–72 vol.3. doi:10.1109/ICASSP.2004.1326558.

[22] R. H.264, Rede h.264 sbtvd, available at: http://www.lapsi.eletro.ufrgs.br/h264/wiki/tiki-index.php (2009).

[23] Ginga, Ginga (2011).
URL http://www.ginga.org.br