Implementation Techniques for Supporting AOSD Methodology

Abstract

The use of a careful domain engineering is essential to achieve the level of portability and efficiency demanded by embedded systems. The Application Oriented System Design (AOSD) methodology guides the development of application-oriented embedded systems from domain analysis to implementation, and uses several software engineering and implementation techniques to achieve this goal. This paper presents some implementation techniques used to support the AOSD methodology.

1. Introduction

Analysis of commonality and variability and proper design and implementation processes in software (and hardware) development is essential for quality, reuse, maintainability and evolution of systems.

Several methods and techniques were created to deal with commonalities and variability of software, including domain analysis, collaboration-based design, family-based design, component-based design and aspect oriented programming.

Our group has worked with the application oriented system design (AOSD), which was proposed to solve some problems that limit the development of embedded systems. This methodology uses concepts of all those other methods and is supported by special implementation techniques and a software environment to assist the design of embedded system instances. This paper presents some implementation techniques used to support the AOSD methodology.

The rest of this paper is organized as follow: section 2 summarizes some well-known methods and techniques that deal with software variability, converging to the AOSD methodology, that is the focus of this paper. Section 3 shows how some of those methods and techniques are implemented in a software environment for supporting the AOSD methodology to generate embedded systems. Section 4 presents some results and conclusions from our experience.

2. Theoretical References

Domain Analysis (DA) is the systems engineering of a family of systems in an application domain though development and application of reusable assets. The goal of DA is to develop a reuse library asset that will be used in the implementation of system instances in the domain family [1]. These assets usually include software components, interface specifications, documentation, test plans and generators. The use of a careful domain analysis is essential to achieve the level of portability and reuse demanded by embedded systems. The domain engineering process consists of systematic development of a domain model and its implementation. A domain model is the representation of common and variant aspects of a number of representative systems of a domain and the rationale for variations.

Object-Oriented Design (OOD) is actually one of the most used method of software design. It has been evolving for more than 20 years and several languages and design tools support it. OOD identify objects with well-defined behavior that enclosures its own data and communicate with other objects though message passing (*function members*), and uses a notation for static and dynamic models [2]. Objects are decomposed and common behaviors form *classes*. Variability is modeled as *subclasses*, or specialized classes. OOD tries to form classes with a single objective (highly coherence) and with a minimum dependence of other classes (low coupling).

Collaboration-Based Design extends OOD to express that an object can play different roles in a system, and that a collaboration can be a better unit of reuse and composition than a class [3], and has the potentiality to guide the development of reusable components. A *collaboration* is defined by a set of objects and an interaction protocol that specifies the roles of each object in the collaboration. The way to implement roles is not specified by the methodology, but it could be performed by using *parametrized classes* (class templates).

Family-Based Design (FBD) was first introduced by Parnas [4]. FBD tries to identify *commonality* and *variability* over application systems. The basic criterion to group functionalities is the commonality. Entities that share common functionalities are group together to form *families* of components. The variability over entities is a family modeled as different *component members* of such family. A new application instance can be composed or customized by selecting the appropriated members of the identified families.

Component-Based Design (CBD) has became one of the most promising approaches to the development of reusable software for embedded systems. Reusability, however, does not simply emerge from components. Components must be designed to be reusable. In this way, components should represent significant entities in the domain they are applied to, which demands the use of other techniques to identify commonalities and variabilities. CBD is also the most promising technique for hardware design as well. Hardware is designed as a composition of reusable components, called *Intellectual Properties* (IP), since it reduces the complexity and design time [5]. IP are usually described in a Hardware Description Language (HLD) or even higher level languages,as C++ [6] or SystemC [7], and transparently translated to an HDL and then interconnected in an on-chip-bus or in a network-on-a-chip. Recent works demonstrated the applicability of other software development techniques to hardware components as well, such as aspect-oriented programming.

Aspect-Oriented Programming (AOP) was introduced by Kiczales [8] to deal with *non-functional properties* of component-based systems., such as security, synchronization, sharing, timing and atomicity. AOP capture non-functional properties in reusable units called *aspects*. Aspects are specified in aspect-oriented languages (eg aspect-java, aspect-c++) and woven with components using *aspect weavers* to generate the system. Although AOP suggests means to adapt components according to an aspect, AOP itself doesn't enforce a design policy that yields aspect independent components.

Application Oriented System Design was proposed by Fröhlich [9] to guide the development of application-oriented operating systems from domain analysis to implementation. It proposes strategies to define components that represent significant entities in different domains. AOSD allows the modeling of independent *abstractions* and organizes them as family members, as defined in the FBD. To reduce environment dependences and to increase abstractions re-usability, AOSD aggregates the aspects separation (from AOP) to the decomposition process. With the use of this concern, it is possible to identify scenario variations and non-functional properties and to model them as *scenario aspects* that crosscut the entire system. The integrated utilization of these and other advanced software engineering techniques allows the development of efficient methodologies for embedded systems design, both in basic software and in hardware domains.

AOSD dictates that scenario dependencies must be factored out as *aspects*, thus keeping abstractions scenario-independent. However, means must be provided to apply factored aspects to abstractions in a transparent and efficient way. The traditional approach to do this would be deploying an *aspect weaver*, though the scenario adapter construct has the same potentialities without requiring an external tool. A *scenario adapter* wraps an abstraction, intermediating its communication with scenario-dependent clients to perform the necessary scenario adaptations.

Inflated interfaces summarize the features of all members of a family, creating an unique view of the family as a "super component". They allow application programmers to write their applications based on well-know, comprehensive interfaces, postponing the decision about which member of the family shall be used until enough configuration knowledge is acquired. The binding of an inflated interface to one of the members of a family can thus be made by automatic configuration tools that identify which features of the family were used in order to choose the simplest/cheaper realization that implements the requested interface subset.

Summarizing, in AOSD, during domain decomposition, abstractions are identified from domain entities (using DA) and grouped in families according to their commonalities. Yet, during this phase, aspect separation is used to shape scenario-independent abstractions, thus enabling them to be reused in a variety of scenarios. These abstractions are subsequently implemented to give rise to the actual software components. This concept also enables an application-oriented embedded system to be automatically generated of out of a set of software and hardware components, since *inflated interfaces* serve as a kind of requirement specification for the system that must be generated.

Embedded Parallel Operating System (EPOS) is one of the first practical strategies using AOSD. EPOS is a framework conceived through AOSD that combines concerns of DA, FBD, OAP, OOD and Static Meta Programming (SMP). Besides operating system components, it has been extended to deal with hardware [10], allowing for the design of hybrid components whose software/hardware implementations are suitable. This approach has so far enabled the development of run-time support systems with architectures that are defined according to the particular needs of applications as a system-on-a-chip (SoC). Indeed, with all these features it seems a promising approach to help solving the problems that currently limit efficiency in embedded system development. The effectiveness of AOSD and EPOS have already been demonstrated in [10], [11], [12], and others.

3. Development

In this section we show how some design methods were implemented in EPOS, present some specific interesting cases and briefly describe new features of a software environment for supporting AOSD.

Figure 1 presents the organization of the component families in EPOS. Every architecture-dependent hardware unit was abstracted as a *hardware mediator*. These constructs are responsible for exporting, through their interfaces, all the functionality needed by higher level *system abstractions*, which are responsible for implementing traditional operating systems services such as memory management, process management, inter-process communication, etc.

Our approach relies on a static configuration mechanism that allows the generation of optimized versions of the operating system and hardware platforms for each of the applications that are going to use it. This approach was implemented using EPOS framework and consists on a repository of hardware and software components, files to represent dependences over components, composition rules, *scenario adapters*, traits and features, and a software environment that use all these stuff to configure and generate application-oriented embedded systems.

Scenario Adapters were conceived in AOSD as a mechanism for AOP without the use of *code weavers*. Scenario adapters are software artifacts that allow intercepting messages to *system abstractions* and the activation of a set of *aspects* that define a *scenario*, and their basic organization is depicted in figure 2. It looks like the *Adapter* design pattern, but there are some fundamental differences. In *scenario adapters*, the `Abstraction` and `Scenario` classes are designed to be bound at compile-time, generating no overhead due to a polymorphic implementation. Even efficient for their purposes, *scenario adapters* do not present the features of correction and code changing presented by code weavers.

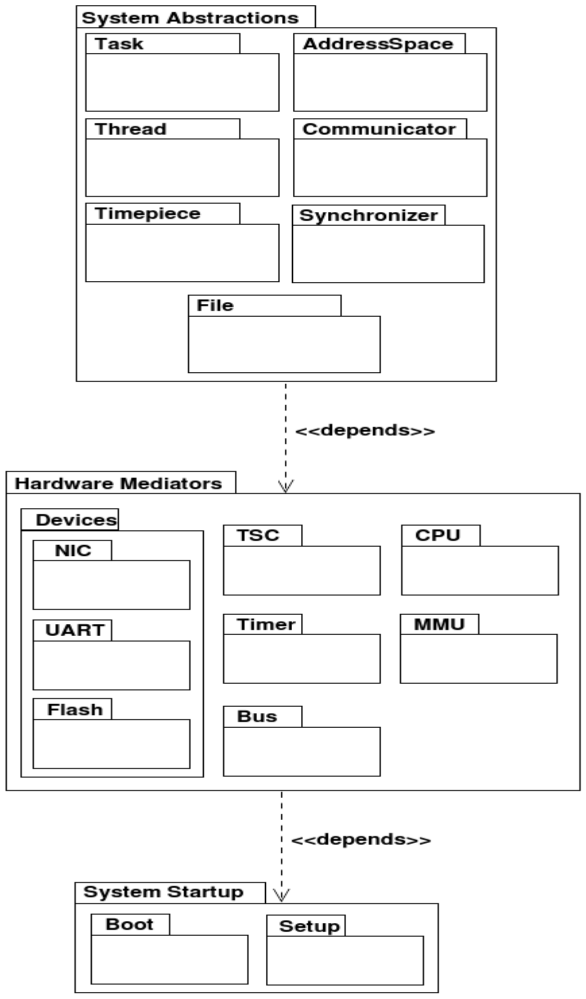


Figure 1: EPOS Domain Model

Figure 3 depicts *scenario adapters* as implemented in EPOS. In this implementation, the `Client` of an `Abstraction` can access it only thought a `Scenario Adapter`. This access is performed indirectly thought a parametrized class `Handle`, which exports the interface of the `Abstraction`. The `Handler` propagates the messages destined for the `Abstraction` to the `Adapter`, which combines the `Scenario` with the `Abstraction` itself, as in figure 2. The `Adapter` intercepts the messages for the `Abstraction` and invokes methods `enter()` and `leave()` enclosuring such messages. This 'interception' is solved in compile-time. This mechanism is extended to support remote method invocation in a structure similar to the *Bridge* design pattern, represented by classes `Proxy` and `Agent`. The following code fragments show the implementation some the main classes involved with *scenario adapters.*

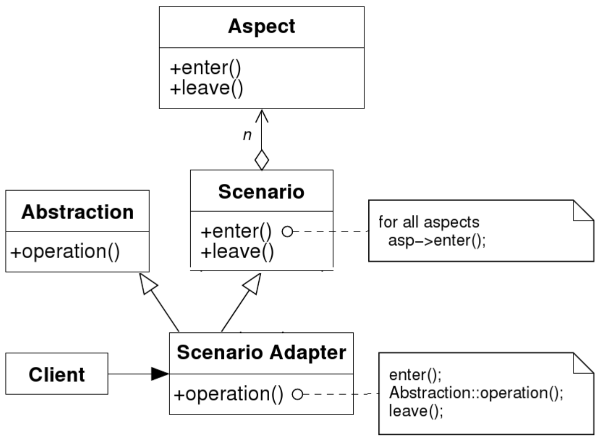


Figure 2: Organization of a *scenario adapter*

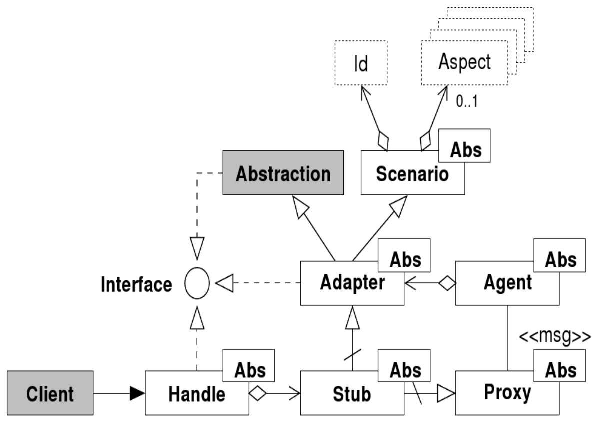


Figure 3: Scenario adapters implemented in EPOS

```
template<class Imp>
class Adapter: public Scenario<Imp>, public
Imp {
   void performAction() {
         Scenario<Imp>::enter();
         Imp::performAction();
         Scenario<Imp>::leave();
   }
};
```

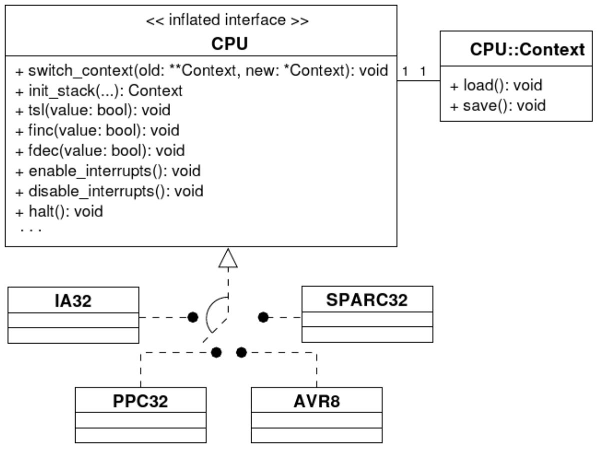
```
template<class Imp>
struct Traits {
   static const bool aspect=false;
};
template<>
struct Traits <Abstraction>{
   static const bool aspect=true;
};
```

```
template<bool active>
class Aspect{
   // Implements an aspect
};
template<>
class Aspect<false>{
   // Implements nothing
};
template<class Imp>
class Scenario: public Aspect
<Traits<Imp>::aspect>,{
   void enter ();
   void leave ();
};
```

Hardware mediators were defined in AOSD as software constructs that mediate the interaction between operating system components and hardware components. They allow system abstractions to be platform independent. Differently from ordinary HALs, hardware mediators do not consist of a monolithic layer: each hardware component is mediated via its own mediator and are organized in families that represent significant entities. The use of static metaprogramming and AOP techniques to implement hardware mediators confer them a significant advantage over VMs and HALs. They are implemented as parametrized classes whose methods are declared inline and defined with embedded assembly instructions. In this way, hardware mediators can even avoid the overhead of function calls.

Other important issue is the variability of hardware platforms, or *machines*. Embedded systems run over different *machines* and therefore demands different hardware mediators to be portable. Specialization, as defined in OOD usually includes the overhead of polymorphism. EPOS defines classes with some abstract methods for each hardware mediator, and each *machine* implements those methods. Special care are taken to guarantee that the correct implementation is statically bound at compile-time, which is done using conditional compiling techniques.

As a simple example of hardware mediator we present some aspects of the CPU mediator. In figure 4 is shown the CPU class, that is the *inflated interface* that summarizes all CPUs features. This class can be implemented by one (and only one) of the specific architectures (IA32, PPC32, SPARC32, AVR8 - event not explicit in figure, EPOS has been ported to other architectures too). The method `finc`, which atomically increments a value, is shown bellow. `finc` may use an atomic assembly instruction, if the architecture has one, or may produce atomicism by disabling and enabling interrupts.



<< inflated interface >>
CPU
+ switch_context(old: **Context, new: *Context): void
+ init_stack(...): Context
+ tsl(value: bool): void
+ finc(value: bool): void
+ fdec(value: bool): void
+ enable_interrupts(): void
+ disable_interrupts(): void
+ halt(): void
...
1 1
CPU::Context
+ load(): void
+ save(): void
IA32
SPARC32
PPC32
AVR8

Figure 4: CPU hardware mediator

```
class CPU_Common { //...
   static int finc(volatile int & number) {
      int old = number;
      number++;
      return old;
   }
}
```

```
class IA32: public CPU_Common {
   //...
   static int finc(volatile int & value) {
      register int old = 1;
      ASMV("lock\n"
        "xadd %0, %2" : "=a"(old) : "a"(old),
"m"(value)); return old;
   }
}
```

```
class AVR8: public CPU_Common {
   //...
   static int finc(volatile int & value) {
      int_disable();
 register bool old = CPU_Common::finc(value);
      int_enable();
      return old;
   }
}
```

The use of configurable hardware as platform for embedded systems, ie, programmable logic devices (PLD) as FPGAs or ASICs, includes other level of variability on the system. Thought Design Space Exploration and Hardware/Software Partitioning techniques, system functionalities can be mapped into software or hardware. In EPOS, the basic element to be mapped is a family member component. Components that can be mapped onto hardware or software are called ***hybrid components***. Figure 5 depicts the organization of *hybrid components.* Each *hybrid component* aggregates a hardware mediator that interfaces several hardware and software components. The main challenge is to design the *hardware mediators* to construct a repository of components with interfaces that are exactly the same independently if the component is mapped onto software or hardware.

A software environment supporting AOSD searchs the repository for *hybrid components* (lookink into XML files that represent dependences over components) and performs hardware/software partitioning based on the final system costs. As the final result of this process, the software environment adjusts a value (HwSw`_Impl_Member`) in a configuration file, for each *hybrid component*, selecting a specific member implementation. Hybrid components are then implemented as parametrized classes.

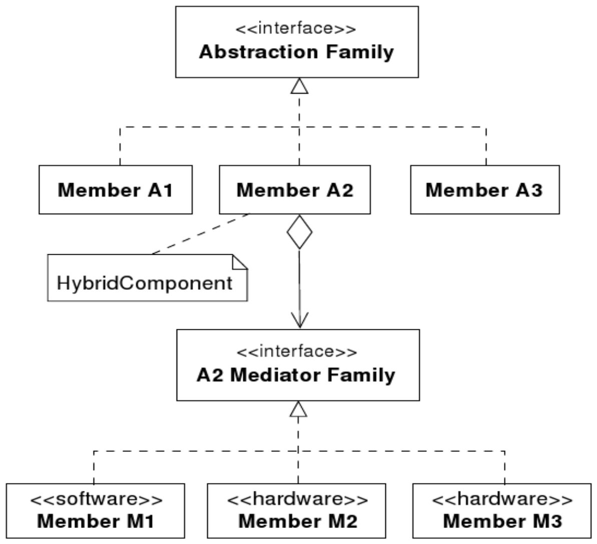


Figure 5. Hybrid hardware/software component organization

The following code fragments show the example of how the *semaphore* abstraction is implemented. The first frame shows the configuration file and the value adjusted by the software environment as result of the partitioning process. In this case, the member 2, which is fully implemented in hardware, was selected. Second frame shows the parametrized class `Semaphore_Imp` and the implementation of its `v()` method in software (`<1>`). Note this implementation uses the `finc()` method of CPU hardware mediator, presented earlier in this paper. The third frame shows the implementation of the mediator for semaphore in hardware (written in VHDL and not presented here). Note the `v()` method now accesses memory mapped registers of a hardware peripheral: the semaphore in hardware. The last frame shows how the *semaphore* abstraction is statically bound to a `Semaphore_Imp` class, using metaprogramming. No overhead is inserted in this process.

```
//..
template <> struct Traits<Synchonizer> {
   static const int HwSw_Impl_Member = 2;
//full hw
   //...
};
```

```
template <int Impl_Member>
class Semaphore_Imp;

template<>
class Semaphore_Imp<1> : public
Synchronizer_Common { // pure software impl.
  // ...
   void v() {if(finc(_value) < 0) wakeup();}
}
```

```
template<>
class Semaphore_Imp<2> { // pure hardware
   // ...
   void v() {
      unsigned int status;
      Thread * thr;
      *sem_cmd = (0x80000000 | (sem_id<<16));
      status = *sem_cmd;
      thr = (Thread *)*sem_thr;
      if (status& STAT_RESUME) thr->resume();
   }
}
```

```
class Semaphore: public Semaphore_Imp
<Traits<Synchronizer>::HwSw_Impl_Member> {
public:
   Semaphore(int n=1): Semaphore_Imp <Traits
<Synchronizer>::HwSw_Impl_Member>(n){}
};
```

Software environment for Supporting AOSD is suit of tools to assist the design of embedded systems using AOSD and EPOS, and it was divided into four major modules: *Analyzer*, *Partitioner*, *Configurator*, and *Generator*.

The *Analyser* is responsible for identifying what features are required from the application, and elaborates a requirement specification that includes methods, types, and constants used by the application. This module seeks the input for references to the components' interfaces (methods that compose the OS API), what could be done based on high level input specifications of the system, such as UML or source-code. The actual implementation of the *Analyzer* assumes the input is the application source-code. It applies a technique that involves the compilation of the application's source code, a look at the resulting object files, and the identification of unresolved symbols (the EPOS API). It's useful to remember the tool modules were designed as independent components. It means that other implementation that reads a XMI file describing the application (with UML diagrams) could also be used to search for references to the components' interfaces and to elaborate a requirement specification, with no modifications to the tool chain. A component dependency tree is produced and used to feed the *Partitioner*. Multiple project alternatives are coded as alternative components (nodes) in such structure.

The description of components must be complete enough so that the *Partitioner* module will be able to automatically identify which abstractions better satisfy the requirements of the application without violating design requirements, generating conflicts or invalid configurations and compositions. A component is defined by a *family* and its set of *members*. In addiction to that, this enriched description can be used to perform design space exploration. A dependency tree with no alternative components corresponds to a unique project alternative and features are used to map how components meet design constraints. The combination of all possible projects, including possible target-platforms, forms the design space to be explored.

The description of the interfaces in a family and its members is the main source of information for the *Configurator*, but correctly assembling a component-based system goes far beyond the verification of syntactic interface conformance: non-functional and behavioral properties must also be conveyed. For this purpose, the component description language includes two special elements: *feature* and *dependency*. These elements can be applied to virtually any other element in the language to specify features provided by components and dependencies among components that cannot be directly deduced from their interfaces. Enriching the description of components with features and dependencies can significantly improve the correctness of the assembly process, helping to avoid inconsistent component arrangements.

In the last module, the *Generator* allows the designer to launch processes that invoke the operating system's makefiles, causing the system instance generation, and processes that invoke synthesis tools that build the hardware platform (if it's a FPGA). Also, the application may be compiled by the *Generator* with parameters that consider the system that was just built for it. Our approach aims at generating real systems, not only simulated ones. Possible implementations of this module could generate a system's model at different abstraction levels (co-simulation models) to provide performance metrics back to the *Partitioner* in an iterative process. A limitation of the actual implementation of the *Generator* is that it only generates the final system, composed by a software image and, depending on target-platform, also the bit stream file to configure the FPGA, but does not simulate the system or obtains performance metrics. Current developments are creating a new *Generator* component to provide such functionality.

4. Results and Conclusions

In this paper we dealt with some problems of developing and implementing embedded systems. We have briefly shown the basic concepts of Application-Oriented System Design methodology that was developed to solve these problems and we have presented some implementation techniques used to support it.

In addition to that we have shown a software environment that assists developers in configuring and generating software and hardware support for embedded systems taking as base a collection of reusable hybrid (hardware and software) components developed according with the Application-Oriented System Design methodology, their dependencies, composition rules and features. The prototype effectively identifies, selects, configures, adapts, and composes those components, generating real and functional embedded systems.

5. References

[1] Comer, E. R. “Domain Analysis: a system approach to software reuse”, Digital Avionics System Conference, 1990, pp. 224-229.

[2] Booch, G. “Object-Oriented Analysis and Design with Applications”, Addison-Wesley, 2sd edition, 1994.

[3] Batory, D., and O'Malley, S. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, ACM Transactions on Software Engineering and Methodology, 1992.

[4] Parnas, D. L. “On the Design and Development of Program Families”, IEEE Transactions on Software Engineering, 1997, pp. 23-26.

[5] Vincentelli, A. S., and Cohn, J. “Platform-based design and software design methodology for embedded systems”, IEEE Design e Test, 2001, pp. 23-33.

[6] de Micheli, G. “Hardware Synthesis from C/C++ Models”, Design, Automation and Test in Europe, 1999.

[7] Open SystemC Initiative. “SystemC Documentation”, Open SystemC Initiative, 2008.

[8] Kiczales, G., Lamping, J. Mendhekar, A. et al. “Aspect-Oriented Programming”, Europen Conference on Object-Oriented Programming, volume 1241 of Lecture Notes in Computer Science, 1997, pp. 220-242.

[9] Fröhlich, A. A. M. “Application-Oriented Operating Systems”, Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001.

[10] Polpeta, F., and Fröhlich, A. A. M. “Hardware Mediators: a Portability Artifact for Component-Based Systems”, International Conference on Embedded and Ubiquitous Computing, volume 3207 of Lecture Notes in Computer Science, 2004, pp. 271-280.

[11] Tondello, G. F., and Fröhlich, A. A. M. “On the Automatic Configuration of Application-Oriented Operating Systems”, 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005.

[12] Schulter, A., Cancian, R. Stemmer, M. R, and Fröhlich, A. A. M. “A Tool for Supporting and Automating the Development of Component-based Embedded Systems”, Journal of Object Technology, 6(9):20, 2007.