On the Automatic Generation of SoC-based Embedded Systems [*]

Fauze Valério Polpeta and Antônio Augusto Fröhlich
Federal University of Santa Catarina
UFSC/CTC/LISHA - PO Box 476
88049-900 Florianpolis - SC, Brazil
{fauze,guto}@lisha.ufsc.br

Abstract

The growing complexity of embedded applications has motivated system designers to search for methods and tools that enable the automatic generation of embedded systems. This paper outlines a strategy for generating customized run-time support systems and specific hardware platforms for dedicated applications. Based on the Application-Oriented System Design *methodology, the approached strategy proposes the use of* Hardware Mediators*—an original portability artifact—as the basis for creating IP-based SoCs that match, in association with a run-time support system, the requirements of dedicated applications. The several steps involved in this process are presented in a detailed case study using an experimental application-oriented operating system instance.*

1. Introduction

Embedded systems are becoming more and more complex, yet, there is no room in this extremely competitive sector for development strategies that incur in extended time-to-market. In this context, the *System-on-a-Chip* (SoC) concept comes as a compromise between system complexity and development costs. Furthermore, the advances in programmable logic devices (PLD) are enabling developers to develop complex SoC designs in a short period of time. This positive effect on time-to-market is making PLDs an important technologic alternative for development of embedded systems.

However, getting a SoC out off a set of IPs is not a trivial task and requires an intricate engineering process. Much effort has been invested in tools that assist designers in selecting and configuring IPs and also in generating the necessary glue logic. These tools usually presuppose an standardized interconnect, such as Wishbone, Amba or Coreconnect [17], and thus can be effective. For instance, the CORAL [2] project from IBM uses the concept of *Virtual Design* to provide the designer with simplified IP descriptions that hide many implementation details.

Indeed, some *embedded systems* can be completely implemented in hardware using this approach, but the more complex the application, the greater is the probability it will need some kind of *run-time support system* and an *application program*. This is, after all, the reason why so many groups are concentrating efforts to develop processor soft cores such as Leon2 and OpenRisc [12]. Nevertheless, run-time support systems are often neglected by currently available SoC development methodologies and tools, being mostly restricted to simple processor scheduling routines and the definition of a hardware abstraction layer. The gap between software and hardware gets even bigger when we recall that one of the primary goals of an operating system is to grant the portability of applications, and ordinary operating systems cannot match the dynamism of SoCs.

In this paper we discuss the use of *Hardware Mediators* [18] to enable the automatic generation of SoC-based embedded systems. The deployment of *Application-Oriented System Design* (AOSD) [4] in the context where hardware mediators were originally proposed—*software-hardware interfacing*—fosters this portability artifact to a new perspective on the design of embedded systems. Mediators define a kind of "machine description" that matches, in association with a run-time support system, the requirements of dedicated applications. The following sections describe the basics of the AOSD method, the concepts of hardware mediators and how these mediators can be deployed for the generation of SoC-based embedded systems. Subsequently, in a case study, with the EPOS system [5], an application-oriented operating system that relies on hardware mediators to foster portability and also to enable automatic hardware generation. The paper is closed with a discussion of related works and the author's perspectives.

2. Application-Oriented System Design

Application-Oriented System Design (AOSD) [4] proposes a strategy to proceed the engineering of a domain towards software components. In principle, an application-oriented decomposition of the problem domain can be obtained following the guidelines of *Object-Oriented De-*

[*]This work was partially supported by FINEP (Financiadora de Estudos e Projetos) grant no. 01.04.0903.00.

composition [3]. However, some subtle yet important differences must be considered. First, object-oriented decomposition gathers objects with similar behavior in class hierarchies by applying variability analysis to identify how one entity specializes the other. Besides leading to the famous "fragile base class" problem [13], this policy assumes that specializations of an abstraction (i.e. *subclasses*) are only deployed in presence of their more generic versions (i.e. *superclasses*).

Applying variability analysis in the sense of *Family-Based Design* [16] to produce independently deployable abstractions, modeled as members of a family, can avoid this restriction and improve on application-orientation. Certainly, some family members will still be modeled as specializations of others, as in *Incremental System Design* [9], but this is no longer an imperative rule. For example, instead of modeling connection-oriented as a specialization of connectionless communication (or vice-versa), what would misuse a network that natively operates in the opposite mode, one could model both as autonomous members of a family.

A second important difference between application-oriented and object-oriented decomposition concerns environmental dependencies. Variability analysis, as carried out in object-oriented decomposition, does not emphasizes the differentiation of variations that belong to the essence of an abstraction from those that emanate from the execution scenarios being considered for it. Abstractions that incorporate environmental dependencies have a smaller chance of being reused in new scenarios, and, given that an application-oriented operating system will be confronted with a new scenario virtually every time a new application is defined, allowing such dependencies could severely hamper the system.

Nevertheless, one can reduce such dependencies by applying the key concept of *Aspect-Oriented Programming* [10], i.e. aspect separation, to the decomposition process. By doing so, one can tell variations that will shape new family members from those that will yield scenario aspects. For example, instead of modeling a new member for a family of communication mechanisms that is able to operate in the presence of multiple threads, one could model multithreading as a scenario aspect that, when activated, would lock the communication mechanism (or some of its operations) in a critical section.

Based on these premises, Application-Oriented Systems Design guides a domain engineering procedure (see Figure 1) that models software components with the aid of three major constructs: families of scenario-independent abstractions, scenario adapters and inflated interfaces.

Families of scenario independent abstractions

During domain decomposition, abstractions are identified from domain entities 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.

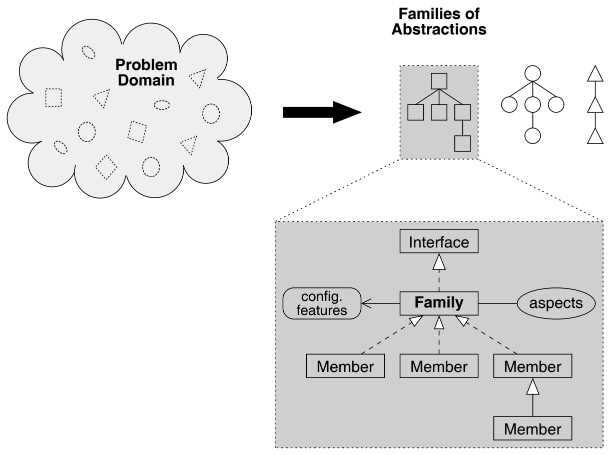

Figure 1. Overview of application-oriented domain decomposition.

Scenario adapters

As explained earlier, AOSD dictates that scenario dependencies must be factored out as *aspects*, thus keeping abstractions scenario-independent. However, for this strategy to work, means must be provided to apply aspects to abstractions in a transparent way. The traditional approach to do this would be deploying an *aspect weaver*, though the *scenario adapter* construct [6] 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 adaptations.

Inflated interfaces

Inflated interfaces summarize the features of all members of a family, creating a unique view of the family as if it were "super component". It allows 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 in the family can thus be performed by automatic configuration tools that identify which features of the family were used in order to choose the simplest realization that implements the requested interface subset at compile-time.

3. Hardware Mediators

An operating system designed according to the premises of Application-Oriented System Design can be summarily viewed as sets of software components that can be configured, adapted and integrated in order to give rise to highly customized and scenario-specific instances of

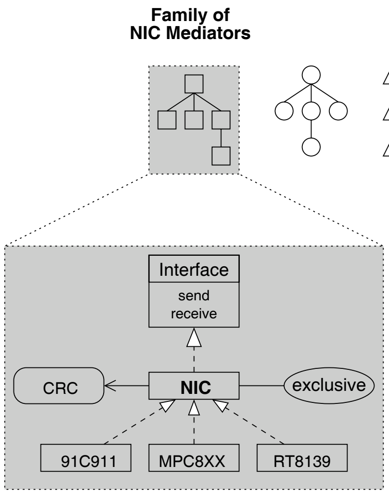

Figure 2. A family of hardware mediators for Network Interface Cards.

run-time support systems. However, besides all the benefits claimed by software component engineering, such a class of run-time support systems is prone to the same need for portability as their more conventional relatives.

Traditional portability strategies, mainly focused in hardware abstraction layers (HAL) and virtual machines (VM), are not concerned with the AOSD's purposes. Being a product of a system engineering process (instead of a domain engineering process), these strategies usually build a monolithic abstraction layer that encapsulates all the resources available in the hardware platform without properly regarding the application needs. Such modeling may be a problem when the platforms to be interfaced are based on SoCs. The diversity of architectures and devices in these platforms lead us to diagnose that the traditional specification techniques for sw-hw interfacing are still far from the ideal "plug-and-play" [15]. In addition, whenever SoCs are built on *Programmable Logic Devices* such as FPGAs, the hardware specifications can be modified in a short period of time [19], and thus compromising much more the portability of the system.

In order to deal with this dynamism and to foster the portability of system abstractions to virtually any architecture, a system designed according to the concepts of AOSD relies on the hardware mediator construct [18]. The main idea behind this portability artifact is not to build an universal hardware abstraction layer or virtual machine, but sustaining an *interface contract* between the operating system and the hardware. Each hardware component is mediated via its own mediator, thus granting the portability of abstractions that use it without creating unnecessary dependencies. Indeed, hardware mediators are intended to be mostly static-metaprogramed and thus "dissolve" themselves in the system abstractions as soon as the interface contract is met. In other words, a hardware mediator delivers the functionality of the corresponding hardware component through a system-oriented interface.

An important element of hardware mediators are *configurable features*, which designate features of mediators that can be switched on and off according to the requirements dictated by abstractions. A configurable feature is not restricted to a flag indicating whether a preexisting hardware feature must be activated or not. Usually, it also incorporates a *Generic Programmed* [14] implementation of the algorithms and data structures that are necessary to implement that feature when the hardware itself does not provide it. An example of configurable feature is the generation of CRC codes in mediators that abstract communication devices.

Likewise abstractions in AOSD, hardware mediators are organized in families whose members represent significant entities in the hardware domain. For instance, a family of Network Interface Cards (`NIC`) mediators could feature members suchs as `91C911`, `MPC8XX`, and `RT8139` (see Figure 2). Such modeling enables the generation of object-code only for those mediators that are necessary to support the application. Non-functional aspects and cross-cutting properties are encapsulated as *scenario aspects* that can be applied to family members as required. For instance, families like `UART` and `NIC` must often operate in exclusive-access mode. This could be achieved by applying a share-control aspect to the families.

4. Generating SoCs Using HW Mediators

Although originally devised to give rise to highly adaptable system-hardware interfaces, hardware mediators can be also used to generate application-oriented hardware instances. More specifically, in the context of programmable logic, where hardware components are usually implemented using hardware description languages (e.g. VHDL, VERILOG). By associating hardware mediators with these decriptions, namely soft-IPs, one can infer which hardware components are necessary to build-up the target hardware. In other words, as soon as a hardware mediator is selected to interface a hardware component, the associated IP is selected from a repository in order to integrate a hardware description that will be synthesized in a PLD. Such a description in the form of a SoC would embed only the hardware components necessary to support the run-time system and, in turn, the application.

Figure 3 illustrates the idea of using hardware mediators to infer hardware components. The SoC generation process consists of selecting the IPs that are associated with the mediators that will be instantiated to support a given application. Such IPs represent significant entities from the hardware domain that are grouped in a repository as families of hardware components. As mentioned, each family of hardware components is associated to a family of mediators whose members are responsible for mediating the hardware elements that the IPs give rise. The information related to this association is held by the *Info* database, which, in turn, provides the information neces-

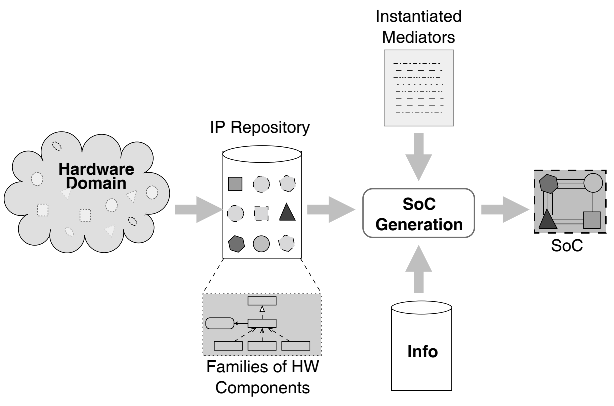

Figure 3. The use of hardware mediators to infer the IPs that will be synthesized.

sary to configure the IPs.

For instance, reconsider the Figure 2 presented in Section 3. As explained earlier, this figure depicts a family of `NIC` hardware mediators. Consider now that it also represents a family of `NIC` hardware components whose members are associated to the members of the respective mediator's family. From a given application that uses a system abstraction to implement an Ethernet network one can infer that a member of the family `NIC` will be instantiated. However, the decision of which specific member will be instantiated, since all members are functionally equivalent, is up to the application's programmer. This situation characterizes what we named a *combined IP-selection*. The hardware devices are induced considering an application requirement and a specific decision of the application's programmer.

Another scenario, named *discreet IP-selection*, is related to the selection of hardware components considering only the application's requirements — no explicit programmer decision must be taken. A good example for this scenario is related to the memory management scheme that will be implemented by the run-time support system. If an application programmer uses system abstractions that depend on *paged* scheme, the `MMU` mediator is automatically induced and, consequently, a memory management unit will be selected for synthesis. Conversely, if a *flat* memory scheme is used no memory management unit is synthesized.

A third scenario, named *explicit IP-selection*, represents the chance of the programmer to choose the hardware components that will be instantiated in the system independently. Even if the respective mediators are "hidden" by system abstractions, the programmer explicitly selects the hardware components that he is intending to embed in the SoC. Indeed, this selection strategy is always taken when the programmer initially specifies which architecture model (e.g. `SPARCV8`, `OR32`) the system will follow.

Furthermore, the approach is not restricted to the specification of which IPs will be instantiated in a PLD. The element *configurable feature* explained earlier can be also deployed on hardware components. It can be used to configure IPs in order to properly support the application and also to switch on and off some functionalities that can be implemented in hardware or in software [8]. As exemplified earlier with CRC, a configurable feature can be used to enable the generation of these codes in an `NIC` mediator for data transmitted over a network. Such codes, otherwise, could be generated by the hardware itself instead of the software. In this case, since the IP that implements the `NIC` device supports generation of CRC codes, a hardware configurable feature would be activated in order to enable the synthesis of these functionalities in the SoC.

Some interesting examples about the deployment of configurable features on hardware components are the dimensioning of the `CPU` cache sizes in order to minimize the time taken to access data in memory [22] and the exploration of scalable on CPU cores. For instance, the *Leon2 processor* [7], which implements a SoC based on the `SPARCV8` [20] architecture, allows the user to specify the number of register windows that will be synthesized into the `CPU` core. Through a configurable feature, one could set up the optimum number of register windows for a given application.

However, it is important to remember that the role played by configurable features is not only the configuration of system components. Considering that a configurable feature is set in order to fulfill a specific application requirement, only those components that implement such a feature can be selected to fulfill this specific requirement. Consequently, configurable features are also deployed in the component selection process, since they guide the identification of family members that are able to support the application. For instance, a *Network_Communicator* system abstraction could implement a *data_integrity* configurable feature that, when activated, would enable the deployment of a data integrity algorithm on transmissions over a network. The main requisite of this feature, in turn, would be an *NIC* mediator or an *NIC* IP that features a data integrity mechanism such as CRC codes.

The Figure 4 presents a diagram of the system's component inferring process. The vertical lines represent the different entity domains of the system. The rows show the three scenarios for IP selection, i.e. *explicit*, *combined* and *discreet*. As illustrated, the configurable features are deployable on operating system abstractions, hardware mediators and hardware components. The gray area in the figure represents functionalities that for one application could be implemented by the hardware mediators and, for other, by the hardware itself.

5. Case Study: SoCs in EPOS

The Embedded Parallel Operating System (EPOS) aims at delivering adequate run-time support for dedicated

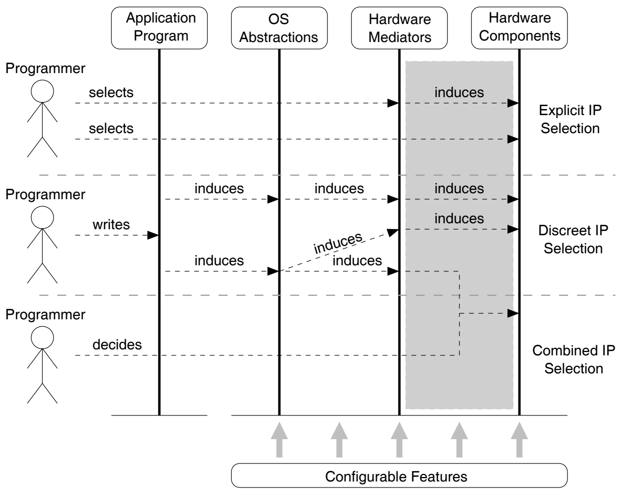

Figure 4. The inferring process of system components.

computing systems. An outcome of *Application-Oriented System Design* method, EPOS consists of families of software components that can be adapted to fulfill the requirements of particular applications. In order to maintain the portability of its systems abstractions and to enable the generation of application-oriented SOCs, the EPOS system relies on the hardware mediator construct.

An application written for EPOS can be submitted to a tool that analyzes it searching for references to the inflated interfaces (see Section 2), thus rendering the features of each family that are necessary to support the application at run-time. This task is accomplished by a tool, the analyzer, that outputs an specification of requirements in the form of partial component interface declarations, including methods, types and constants that were used by the application.

The primary specification produced by the analyzer is subsequently fed into a second tool, the configurator, that consults a build-up database to create the description of the system's configuration [21]. This database holds information about each component in the repository, as well as dependencies and composition rules that are used by the configurator to build a dependency tree. The output of the configurator consists of a set of keys that define the binding of inflated interfaces to abstractions and activate the scenario aspects eventually identified as necessary to satisfy the constraints dictated by the target application or by the configured execution scenario. On the side of hardware components, the configurator produces a list of instantiated mediators and specifies which of these mediators are associated to IPs.

The last step in the generation process is accomplished by the generator. This tool translates the keys produced by the configurator into parameters for a statically metaprogramed component framework and causes the compilation of a tailored operating system instance. In addition, whenever a SOC needs to be tailored, the `generator` produces a synthesis configuration file that holds the parameters for configuring the IPs and the information necessary to glue the IPs in a SOC. Written in a hardware description language, this configuration file and the selected IPs are passed to a third-party tool, which performs the translation of the SOC specification into *netlists*. Following, by considering the target PLD technology, these netlists are finally translated into a configuration *bitstream*. An overview of the whole procedure is depicted in Figure 5

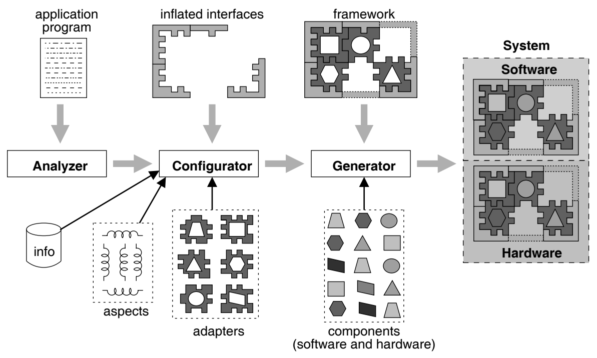

Figure 5. An overview of the tools involved in the automatic generation of run-time support and hardware instances.

5.1. Sample Instance

In order to evaluate our approach for automating the design of SoC-based embedded systems we used the *Leon2 Processor*. The "modular" design of LEON2 enable us to specify which of its IPs will be synthesized in the SOC. The logic necessary to glue the IPs is implicitly defined in the source-code through a set of programming asserts, which are, in turn, used to properly configure and plug the IPs to an AMBA bus framework [1]. The Figure 6 shows the block diagram of LEON2. Besides the CPU core, LEON2 features a set of peripherals that can be included in the SOC according to the final application.

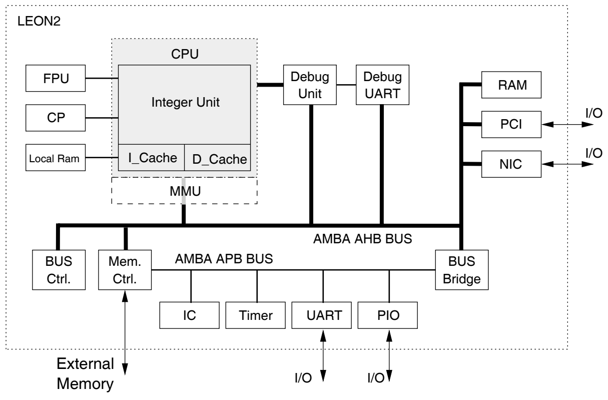

Figure 6. Block diagram of Leon2.

The experiments were performed on a Virtex2 FPGA development kit [24] and consisted in evaluating an appli-

```
char * buffer;
Semaphore empty(BUF_SIZE), full(0);
Serial_Communicator COM;

int consumer() {
   int count_c, out = count_c = 0;
   while (count_c < COUNT) {
      full.p();
      // deal with data in buffer[out]
      COM->send(buffer,1);
      out = (out + 1) % BUF_SIZE;
      empty.v();
      count_c++;
   }
}

int main() {

   Thread * cons = new Thread(&consumer);
   buffer = new char[BUF_SIZE];

   // producer
   int count_p, in = count_p = 0;
   while (count_p < COUNT) {
      empty.p();
      COM->receive(buffer,1);
      // deal with data in buffer[in]
      in = (in + 1) % BUF_SIZE;
      full.v();
      count_p++;
   }
}
```

Figure 7. A source code fragment of the experimental application.

cation for which a run-time support system and a SoC would be generated. The application implemented two threads, *Producer* and *Consumer*, which were executed in a concurrent environment in order to send and receive data stored in a shared *bounded buffer* through an UART (see Figure 7). As regards the memory management scheme, a *flat* address-space was used and therefore, no MMU IP was instantiated in the system. Furthermore, in order to have some integrity control on the transmitted data, the configurable feature *data_integrity* was enabled on the *Serial_Communicator* abstraction, which, in turn, provides the programmer with a high-level communication system interface. [1]

```
static void int_handler(int i, Function * h) {
  // interrupt_vector[i] = h;
}

template <Function * isr>
static void handler_wrapper() {
  // context saving
  // call isr()
  // context restoring
}
```

Figure 8. A source code fragment of the IC mediator in EPOS.

[1]The mutual exclusion concerning the Serial_Communicator is reached through the deployment of scenario adapters (see Section 2).

Aiming at signaling the *Producer* thread to deal with new data in the UART buffer and also for thread scheduling, the mechanism of *interrupts* was used. Consequently, the mediator and the IP of the Interrupt Controller (IC) were selected to be instantiated. Figure 8 shows a source-code fragment of the IC mediator in EPOS. A template-function named *handler_wrapper* is used to encapsulate architectural-specific operations that are performed before and after calling an ISR routine. Such approach enable us to implement architectural-independent ISRs, which are, in turn, registered by the IC mediator through the function *int_handler*. For instance, the registering of the *uart_isr* in this experimental application is accomplished by the following statement:

```
IC :: int_handler ( IC :: INT_UART , &IC:: handler_wrapper
  <uart_isr > );
```

Figure 9 depicts the block diagrams of the SoC that was generated after submiting the application to the sequence of tools presented in Section 5.

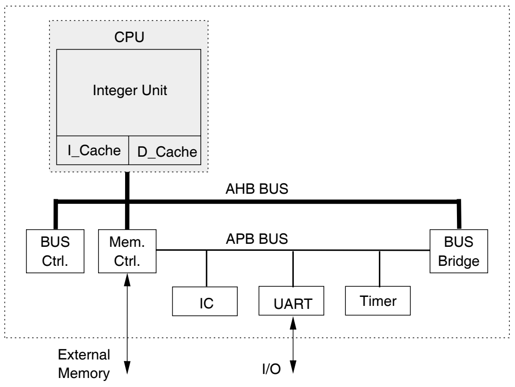

Figure 9. The block diagram of the SoC generated for the experimental application.

In order to illustrate the process of inferring system components in this experimental application, Figure 10 presents the selection decisions that were made by the programmer and by the EPOS configuration environment in order to identify the software and hardware components that better fulfill the application's requirements. The inferring process starts with the selection of which architecture model the system will follow. Subsequently, based on references to the *inflated interfaces*, we infer the operating system abstractions that will support the communication system, i.e. the *Serial_Communicator*, *Concurrent_Thread*, *Semaphore* and *Flat_Addr_Space*. Following, by considering which hardware mediators these abstractions rely on, the hardware components are also inferred, in this case, the *leon2_uart*, *leon2_ic* and *leon2_timer*. Additionally, notice that the enabling of the configurable feature *data_integrity* on the *Serial_Communicator* induces the activation of the integrity mechanism *parity_check* in the UART device. Further, this mechanism is configured in order to specify which type of parity control will be used: *odd* or *even*.

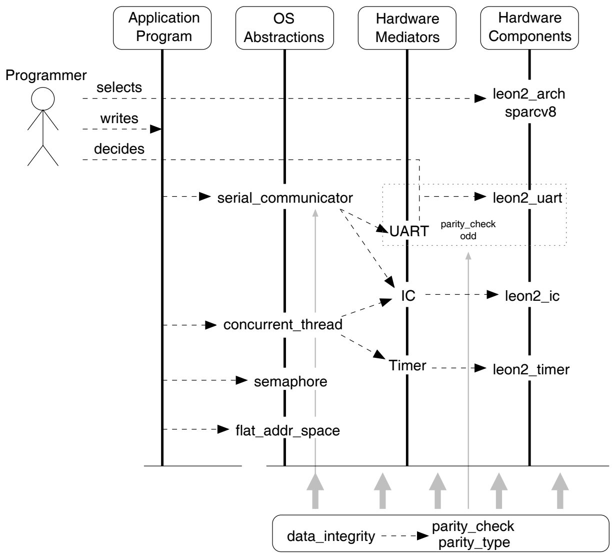

Figure 10. The component inferring flow of the experimental application.

Aiming at clarifying the expressiveness of this sample instance, it is important to compare the obtained results to the numbers of ordinary operating system that were ported to the LEON2 platform. Usually these systems are generated to compromise all the features that the SOC is able to provide. The absence of a component-based engineering and the lack of modern software engineering techniques affects not only the size and performance of the final system, but also increases NRE costs and time-to-market. Table 1 presents the size of the EPOS run-time support system that was generated to support this experimental application and the sizes of two traditional ports of UCLINUX [23] and ECOS [11], also customized to support this same experimental application on the LEON2 platform. Additionally, some statistics related to the size of the SOC generated on this sample instance and the size of LEON2 when it was synthesized with all of its implemented devices are showed in table 2. [2]

OS	.text (bytes)	.data (bytes)	.bss (bytes)
EPOS	8,988	28	8,400
eCos	17,152	796	34,040
uClinux	840,712	44,700	72,649

Table 1. The code segment sizes of EPOS, uCLinux and eCos system images to Leon2.

In respect to the ECOS system, which is based on a component-based approach, it is possible to perform a fine selection of the system abstractions that the programmer wants to provides in the run-time support system. However eCos does not make use of *aspect-oriented programming* and consequently the number of system abstractions in ECOS may be as large as the number of application scenarios that we can in embedded systems. Such modeling makes the component selection process difficult, since ECOS has no automatic component inferring mechanism. Another shortcoming of both, ECOS and UCLINUX, is the hardware abstraction layer portability strategy. Thus, as soon as the programmer selects the platform that will support the application, the HAL is entirely imputed in the system, increasing the size of the run-time support system. For instance, the eCos's HAL for the LEON2 platform always presupposes an `UART` device.

[2]Because of some Leon2's design restrictions the co-processor (CP) and the PCI were not included.

SoC	Size (LUTs)	Virtex2 FPGA Area
Custom	6,792	31,0 %
Full	14,582	67,0 %

Table 2. The sizes of the application-oriented Leon2 instance generated in EPOS and the fully synthesized Leon2 SoC.

The presented application was also evaluated in terms of performance in EPOS and ECOS. The system instances were experimented under same conditions in a Leon2 SoC running at 54 Mhz with a physical loop-back plugged to the serial I/O connector in order to use the same port for data relaying. The application execution time was measured after 16,000 send-receive operations. The obtained results are showed in Table 3. Notice that these numbers do not reflect the transmission times, which represent approximately 16 seconds in total when the serial transmission baud-rate is limited to 38,400 bps.

Operating System	Time taken (ms)
EPOS	45.42
eCos	132.67

Table 3. The time taken to execute the experimental application in EPOS and eCos.

6. Conclusions

In this article we conjectured about the automatic generation of SoC-based embedded systems. The deployment of AOSD in the context where hardware mediators were originally proposed leaded us to use this portability artifact to give rise to hardware descriptions that match, in association with the run-time support system, the requirements of dedicated applications. We claimed that to deal with the dynamism of SOCs and PLDs, operating systems must rely on refined software-hardware interfaces where each hardware component is abstracted by a single abstraction, in our case a hardware mediator. Furthermore, we extended to soft-cores the premise of only instantiating system abstractions and hardware mediators when they are recognized as application's needs. In this sense, by associating hardware mediators to the Leon2's IPs, we generated in a experimental case study the necessary run-time support system and respective system-on-a-chip for a given application.

The approached strategy does not represent a complete solution to automate the process of generating SoC-based embedded systems, whereas the programmer still takes some decisions. Otherwise we believe that it represents an important step in this direction since we have promoted the generation of run-time support systems and SoC instances in the same generation flow according to a well-defined methodology.

7. Future Work

The presented results are quite simple and they just confirmed the viability of automatically generating run-time support systems and system-on-chip instances considering application's requirements. However, as exemplified in Section 4, we are not only able to identify which devices shall be instantiated in a SoC but, in fact, to configure each component of our system in order to better fit the application's requirements. In this sense a large gamma of new experiments started to be evaluated, such as processor scalability, memory hierarchy exploration and power management. The results obtained so far are encouraging and we hope present them soon.

References

[1] Advanced RISC Machines Limited - ARM. *The de facto Standard for On-Chip Bus*, online document edition, 2003. http://www.arm.com/ products/ solutions/ AMBAHomePage.html.

[2] R. A. Bergamaschi, S. Bhattacharya, R. Wagner, C. Fellenz, M. Muhlada, W. R. Lee, F. White, and J.-M. Daveau. Automating the Design of SOCs Using Cores. *IEEE Des. Test*, 18(5):32–45, 2001.

[3] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[4] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug. 2001.

[5] A. A. Fröhlich and W. Schröder-Preikschat. High Performance Application-oriented Operating Systems – The EPOS Approach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 3–9, Natal, Brazil, Sep. 1999.

[6] A. A. Fröhlich and W. Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[7] Gaisler Research Laboratory. *LEON2 XST User's Manual*, 1.0.22 edition, May 2004.

[8] M. Grünewald, J.-C. Niemann, and U. Rückert. A performance evaluation method for optimizing embedded applications. In *IWSOC2003 The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, Alberta, Canada, June 2003.

[9] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Commun. ACM*, 19(5):266–272, 1976.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[11] A. Massa. *Embedded Software Development with eCos*. Prentice Hall PTR, 1 edition, 2002.

[12] D. Mattsson and M. Christensson. Evaluation of synthesizable CPU cores. Technical report, Chalmers University Of Technology, 2004.

[13] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag, 1998.

[14] D. R. Musser and A. A. Stepanov. Generic Programming. In *Proceedings of the First International Joint' Conference of ISSAC and AAECC*, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.

[15] G. Neville-Neil and T. Whitney. SoC: Software, Hardware, Nightmare, Bliss. *ACM Queue*, 1(2):24, 2003.

[16] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, Mar. 1976.

[17] P. Pelgrims, T. Tierens, and D. Driessens. Overview: Excalibur, LEON, Microblaze, Nios, OpenRisc, VirtexII Pro. Technical report, DE NAYER Instituut, 2003.

[18] F. V. Polpeta and A. A. Fröhlich. Hardware Mediators: a Portability Artifact for Component-Based Systems. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, volume 3207 of *LNCS*, pages 271–280, Aizu,Japan, Aug. 2004. Springer.

[19] R. A. Rutenbar, M. Baron, T. Daniel, R. Jayaraman, Z. Or-Bach, J. Rose, and C. Sechen. (When) Will FPGAs kill ASICs? In *Proceedings of the 38th conference on Design automation*, pages 321–322. ACM Press, 2001.

[20] SPARC Internation Inc. *The SPARC Architecture Manual*, sav080si9308 edition, 1992.

[21] G. F. Tondello and A. A. Fröhlich. On the Automatic Configuration of Application-Oriented Operating Systems. In *3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, Jan. 2005.

[22] P. Viana, E. Barros, R. A. Sandro Rigo, and G. Araujo. Exploring Memory Hierarchy with ArchC. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, São Paulo, Brazil, October 2003. IEEE CS.

[23] M. Wurm. uClinux for Sparc-MMUless with Ethernet MAC. Technical report, Graz University of Technology, 2003. http://www.sbox.tugraz.at/ home/ m/ miwu/ sparc/.

[24] Xilinx Inc. *MicroBlaze and Multimedia Development Board User Guide*, ug020 (v1.0) edition, 2002.