CAN development according to Application-Oriented System Design

Danillo Moura Santos and Antônio Augusto M. Fröhlich
Federal University of Santa Catarina
Laboratory for Software and Hardware Integration
UFSC/CTC/LISHA P.O.BOX 476, Florianópolis, SC, Brazil
{danillo,guto}@lisha.ufsc.br

Rafael Luiz Cancian
Department of Automation and Systems Engineering - DAS
Federal University of Santa Catarina - UFSC
cancian@das.ufsc.br

Abstract

CAN bus system is widely used in cars, for communicating sensors and actuators. FPGA chips are becoming widespread, being employed also in industrial and automotive fields. The embedded systems development process gets more complex with the use of programmable hardware, as FPGAs, in this process the design must address the software and the hardware development. This paper shows the design of a CAN listener application, using AOSD methodology to guide the developer in the software application development and hardware inference based on application code.

Keywords: embedded systems design, IPs, AOSD, CAN

1. Introduction

Controller Area Network (CAN) is a serial data communications bus, for real-time applications, developed originally by Bosch for use in cars. It is especially suited for networking intelligent devices as well as sensors and actuators within a system or sub-system. CAN is increasingly being used in industrial field bus systems, as in automobiles, where the application requirements are: low cost, the ability to function in a difficult electrical environment, a high degree of real-time capability and ease of use. Developers have also used CAN in the field of medical engineering, because they have to meet particularly stringent safety requirements.

In the classical development process of embedded systems, regardless of the use of CAN bus system, a microcontroller is usually the core of the system. Additional integrated circuits (ICs) perform specific functions, such as communication, sensing and actuation. As this process has been used for several decades, existing tools and technologies ease the development process. Many modern micro-controller architectures present a wide series of peripherals (e.g. uart, timers, CAN controllers, etc.) on a single chip. The system designer, according to this process, analyzes the application requirements and search for a suitable architecture that meet them. Results in this design strategy are thus very dependent of the designer's experience.

The use of FPGAs is becoming an attractive alternative for low cost embedded applications, because of its hardware reprogram-ability characteristics and the emergence of low cost devices. These devices allow integrated design of hardware and software systems in a chip. Many FPGA vendors provide IPs (hardware synthesizable components) for many functions, like controllers, sensors and even CPUs. Recent FPGAs chips have even electrical transceivers in the same chip.

The Application-Oriented System Design (AOSD) design methodology guides the designer in the development process, easing the management of high complexity systems. The designer builds the software application and the system is generated semi-automatically. Recently extensions of AOSD [1] allow automatic hardware generation, resulting in software and hardware tailored to the application. The semi-automatic component inference process guides the developer in the selection of components that will compose the final system. In this paper we elaborate on a CAN listener developed according to this strategy concepts, which can be used as a base to further FPGA-based CAN applications. This paper is organized as follows: Section 2 has a short description of AOSD methodology and its hardware generation process. Section 3 presents the CAN bus listener case study. Section 4 shows the analysis of the development. Section 6 concludes and presents currently work under development.

2. AOSD basics

Application-Oriented System Design (AOSD) is a domain engineering methodology that elaborates on the well-known domain decomposition strategies behind Family-Based Design (FBD) [2] and Object-Orientation (OO), i.e. *commonality* and *variability* analysis, to add the concept of *aspect* identification and separation yet at the early stages of design citeFrohlich:2001. In this way, AOSD guides a domain engineering towards families of components, of which execution scenario dependencies are factored out as "aspects" and external relationships are captured in a component framework.

Application-Oriented Systems Design proposes a domain engineering procedure that models software components using three constructs: families of scenario-independent abstractions, scenario adapters and inflated interfaces.

Families of scenario independent abstractions are identified during the domain decomposition phase. Abstractions are identified from the domain entities and grouped in families according to their common characteristics. During this phase, the aspect separation process is also performed. This allows abstractions to be reused in different scenarios. Software components are then implemented according to this abstractions.

Scenario adapters are used to solve scenario dependencies, these 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. This is achieved using scenario adapters [3], that wraps an abstraction, inter-mediating its communication with scenario dependent clients to realize the necessary changes, without overhead.

Inflated interfaces hold the features of all members in a family, resulting in a unique view of the family as if it were a "super component". This allows application developers to write applications with base in a clear and well-known interface, postponing the decision about which member of the family shall be used until the moment the system is generated. The association between an inflated interface and a specific member of the family will automatically be made by the configuration tool, that identify which properties of the family have been used, in order to choose the simplest member of the family that implements the required interface. This member will so be aggregated to the OS in compile-time.

2.1. Interfacing Hardware Components

In order to allow the components and the operating system to be portable to most architectures, a system designed according to AOSD makes use of Hardware Mediators [4]. The main idea of this portability artifact is to keep an *interface contract* between the operating system and the hardware. Each hardware component is accessed through its own mediator, thus granting the portability of abstractions that use it without creating unnecessary dependencies. Mediators are mostly static-metaprogrammed and "dissolve" themselves in the system abstractions as the interface contract is met. In other words, a hardware mediator delivers the functionality of the corresponding hardware component through an operating system oriented interface.

Hardware mediators abstractions have *configurable features*, that allow some hardware components features to be switched on or off, according to the requirements of the abstraction. However, these properties aren't only flags that can be turned on and off. Configurable features may be implemented using generic programming, thus allowing the implementation of structures and algorithms in software without considerable overhead. An example could be the generation of cyclic redundant check (CRC) codes as a configurable feature of a communication hardware component.

2.2. Hardware Inference according to AOSD

Hardware mediators have initially been created to facilitate the portability of systems developed following AOSD methodology. However, because of its straight relation to hardware components, they may also be used to identify the hardware components that are really necessary to build a system that supports a specific application.

In the context of Programmable Logic Devices (PLD) where IPs are usually implemented using hardware description languages like VHDL and Verilog, hardware mediators might infer the IPs really required to the system hardware and some of these IPs properties. These IPs are identified as the hardware mediator is instantiated by the application, thus the system hardware will have only the components required to support the OS and consequently the application.

The IP selection might be done according only to application requirements, when no explicit decisions must be taken by the programmer. This scenario is named *discrete IP-selection*. This approach expects that only a specific IP fulfill the system requirements. An application property determines the inference of one specific IP. A good example of this scenario is the activation of *paged* memory in the operating system, this would infer a memory management unit (MMU) IP with *paged* memory support.

There is also a third scenario named *explicit IP-selection* where the programmer can choose all hardware components that will be instantiated independently. This is useful when a hardware component is hidden by a system abstraction. In this case the programmer still can select the component that should be embed in the system.

The hardware mediators configurable features may be deployed in hardware components. Configurable features direct the inference of specific hardware components. An example is the use of CRC codes by a NIC device, the IP representing this NIC should be capable of generating CRC codes.

3. The CAN bus Listener

The Controller Area Network (CAN) BUS Listener was developed to transform CAN packets into packets to be transmitted through a serial interface (UART). This allows monitoring of CAN buses, making it possible to identify packets and the information they contain. The CAN protocol is a real time, serial, broadcast protocol and has a high security level. CAN packets have an identifier that must be unique in all the network. This identifier states the packet content and its priority. CAN buses are present in most modern vehicles, air-planes, factories and others. Identifying the packets and its contents make it possible to monitor vehicle properties like fuel consumption, break usage, acceleration rate and others.

The CAN Listener application consists of a thread that waits for CAN Packets reception. This thread locks whenever reception is not taking place. After reception, the packet is processed and transformed into a sequence of bytes consisting of two separator bytes, an identifier, up to 8 data bytes, and a CRC code. The packet is then sent through a serial interface, and the application waits for an acknowledgment code. If no acknowledgment is received, the packet is sent once more.

The CAN listener was implemented in a VirtexII-Pro XC2V30 Field Programmable Gate Array (FPGA) present in the ML310 evaluation board from Xilinx. This system used one of the PowerPC 405 hard cores found in this FPGA. This application doesn't require a powerful global purpose processor like this, it could easily be implemented in a 8 bits micro-controller (it was actually first implemented in an AT90CAN128 device, an AVR like micro-controller produced by Atmel), but we decided to use PPC to easy the development, as the processor was already in the FPGA chip and has well documented buses.

Other IPs were inferred from the application. These IPs, required by the application, were also instantiated in the VirtexII-Pro FPGA. The application software was written using EPOS API, EPOS is an Embedded Parallel Operating System developed according to AOSD concepts. The Xilinx CAD tools EDK (Embedded Development Kit) and ISE (Integrated Software Environment) were used to integrate and synthesize these components. The Mentor Graphics Modelsim tool was used to simulate and test the IPs.

The CAN Listener application code uses a UART serial interface mediator to send the serial packets created with the contents of received CAN packets. This UART was *combined selected* to compose the system hardware. A CAN controller IP was necessary to allow the system to communicate with the CAN buses. The CAN controller was inferred straight from application (*discrete selected*), as it directly instantiates CAN hardware mediators to receive CAN packets.

As there was no CAN controller IP available in the EDK development tool, an open-source CAN IP found at [5] was used. This IP is compatible with the Wishbone bus interface [6]. As there is no native support for this bus interface in our development platform, the IP interface had to be adapted to the OPB bus. The IP developers made the interface definition and its logic very dependent, making the development of a new interface to the IP virtually impossible. Thus, a bridge between OPB Wishbone buses interface was created. Because of the differences between the two buses, the bridge development process was very laborious, requiring a complex state machine, and several simulation cycles before the integration to the system.

The CAN Listener application also required access to RAM memory, so a DDR memory controller was *discrete selected* to the system hardware. The ML310 platform has a 256 Mb DDR memory and EDK has a memory controller IP to support this memory access. An interrupt controller was also *discrete selected* to compose the system hardware. Some IPs present in the platform were removed to avoid unnecessary overhead to the system. The PCI Bridge, USB interface controller, the SPI controller, the parallel port controller, among others, were excluded. Figure 1 presents a simplified schematic of the CAN bus Listener hardware.

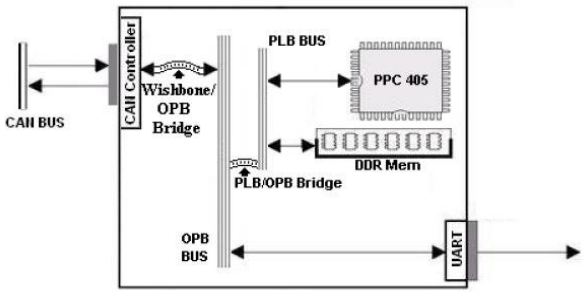

Figure 1. CAN Listener schematic

This case study demonstrated that integration of different IP bus technologies is not trivial, and that the lack of hardware components in the CAD development tools may generate large integration efforts.

4. Case study results

The AOSD methodology has shown itself efficient in the design of embedded system software and hardware. In this case study, we realized that it takes almost the same development time, compared to the development using traditional micro-controller-based strategies. Hardware development based on the application requirements simplified the overall process. The developer knows exactly which hardware components will be necessary to support the application.

The use of EPOS, which is an operating system developed following AOSD concepts, makes the application code extremely efficient. As EPOS is based in software components, only the components necessary to the application functioning will be compiled to object-code. This

results in an efficient code with reduced size, since no unnecessary resources management will be present. Moreover, the use of EPOS enabled the application software to be executed in different architectures. EPOS is a multiplatform operating system, thus the application may be compiled to run in different architectures, as long as this architecture has the hardware components necessary to support the application requirements.

Most of the time, the infer process creates a relation of one hardware mediator to one hardware component. It would be ideal that a hardware component could be tailored to a hardware mediator, instead of using ready-made IPs. The designer should be able to configure an IP according to the hardware mediator being instantiated by the application. This would result in smaller hardware components, extremely adapted to the application.

Hardware components aren't usually developed to be totally configurable, only some of them have some *generic* fields that can be modified by the developer. If hardware components were developed following AOSD approach, every IP would be specifically tailored to the system in which it is being used. This would reduce the size of these components, saving FPGA area.

As seen in the case study, IPs that haven't a bus interface standard similar to the system bus standard requires large efforts for adaptation. In addition to this, OnChip buses bridges creates additional overhead to the system and requires more FPGA area. Many IP cores are initially designed to perform specific functions, evolving to more complex components, which incorporates others functions, thus becoming reusable to others designs. If the IP function is well separated from the IP interface in the design, it should be easier to create new interfaces. Thus, IPs really adaptable might be developed according to AOSD principles and have interfaces independent of logic.

In the traditional EPOS design process, the application developer defines a base architecture and then writes the application. However, in embedded systems development, it is common to choose the cheapest architecture that fulfills the application requirements. Design space explorations might allow application programmers to first write an application, and then choose the cheapest suitable architecture for his design.

5. Conclusion and future work

The FPGA-based embedded system design growing complexity, the wide use of FPGAs in automotive and industrial embedded systems and design strategies deficiencies have motivated this work. Current design strategies have not been able to keep up with the growing complexity of embedded systems design.

Using AOSD strategy and hardware automatic generation, it is possible to infer specific hardware components from the application code, using the artifact of hardware mediators. This reduces a lot the development time and results in a final system with the least number of hardware components possible.

The use of adapted hardware components together with the AOSD methodology may result in a smaller system (in terms of FPGA area). These components would be smaller than off the shelf components. Only what is necessary to meet the requirements of the application will be in the final hardware.

Our present work is the design of hardware components according to AOSD methodology, with interface and logic independent. This allow the generation of systems hardware with the least number of components and these components might be tailored to the application. The development of the IP interface independent of the IP logic makes it easier to reuse the IP in different buses standards.

6. Acknowledgments

We thanks all the LISHA team, especially Lucas for the text revision and suggestions.

References

[1] F. V. Polpeta and A. A. Fröhlich, "On the Automatic Generation of SoC-based Embedded Systems", *In: Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.

[2] D. L. Parnas, "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, vol. 2, pp. 1–9, March 1976.

[3] A. A. Fröhlich and W. Schroder-Preikschat, "Scenario Adapters: Efficiently Adapting Components", *In Proceedings of the 4th Wolrd Multiconference on Systemics, Cybernetics and Informatics*, July 2002.

[4] 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*, vol. 3207, pp. 271–280, 2004.

[5] Opencores, "Opencores", Technical report, Opencores, 2006.

[6] S. Opencores, "WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores", Technical report, Opencores, 2002.

[7] A. A. Fröhlich, *Application-Oriented Operating Systems*, Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 1 edition, 2001.

[8] A. S. Vincentelli and J. Cohn, "Platform-based Design", *IEEE Design e Test*, vol. 18, no. 6, pp. 23 – 33, November 2001.

[9] e. a. Kiczales, Gregor, "Aspect-Oriented Programming", *In Proceedings of the European Conference on Object-oriented Programming*, vol. 1241, pp. 220–242, June 1997.

[10] IBM, "The CoreConnect Bus Architecture", Technical report, IBM, 1999.

[11] Xilinx, "Xilinx LogicCore PLB IPIF (v2.01.a)", Technical report, Xilinx, 2004.

[12] Philips, "Philips Nexperia Platform", Technical report, Philips, 2004.

[13] T. Instrument, "OMAP Texas Instrument Technology", Technical report, Texas Instrument, 2004.