JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 1, No. 1, January–February 2007

A Tool for Supporting and Automating the Development of Component-based Embedded Systems

Rafael L. Cancian, Marcelo R. Stemmer, Departement of Eletrical Engineering, Federal University of Santa Catarina
Alexandre Schulter, Antônio A. M. Fröhlich, Departement of Computer Science, Federal University of Santa Catarina

Embedded systems are comprised of hardware and software and usually run dedicated applications in environments with highly restricted resources, such as memory constrained devices, microcontrollers with low processing power, and wireless sensors running on batteries. These systems must exactly match applications' requirements, with minimum support. The growth in application complexity and even more strong constraints demand new approaches, methodologies, and tools to assist embedded systems development. Usually, in this domain, reuse of components, architectural transparency, low overhead, and reconfigurability are essential features. The Application-Oriented System Design (AOSD) method was created to deal with these issues, and aims at guiding the development of embedded systems that exactly match application requirements. To deliver each application a tailored run-time support system calls for a good combination of object oriented techniques, the separation of functional and non-functional aspects, some implementation techniques, and a sophisticated tool that helps the developer in managing component configurations and automating the generation of embedded systems.
This paper describes a configuration and system generation tool that is being successfully used with EPOS (Embedded and Parallel Operating System), an OS developed using AOSD. This tool receives the application source-code (using EPOS API) as input and, after a few mouse clicks, builds the entire computational support, comprised by software and, if hardware is reconfigurable, the FPGA configuration file. This paper also shows that the design of these tools allows them to be used for configuring and building several other systems, not only EPOS. The development of this tool enabled the configuration and generation of several embedded systems instances for several different architectures in an automatic way. To illustrate this process and the tool's usage, this paper describes a case study of the generation of an embedded system that supports a simple audio decoder application.

1 INTRODUCTION

Embedded systems are being extensively used in several industrial sectors as an effective alternative to control machines, automobiles, domestic appliances, personal

gadgets, and virtually every device that includes electronic components. Recent statistics reveal that over 99% of the microprocessors produced nowadays are used in embedded systems and that in 2005 the number of embedded systems in the planet rose above the number of humans [9]. Besides growing in number, these embedded systems are also becoming more and more complex as they benefit from microchip advances.

In this context, the System-on-a-Chip (SoC) approach emerges as a compromise between complexity and cost. Programmable Logic Devices (PLD) enable developers to evaluate complex designs in short periods of time by applying techniques that are closer to software development than to traditional hardware development. Getting a SoC out off a Field-Programmable Gate Array (FPGA), however, is not a trivial task and requires an intricate engineering process. From the hardware perspective, much effort has been put in tools that assist designers in selecting and configuring hardware components, or Intellectual Property (IP) blocks, and also in generating the necessary glue logic. Indeed, some embedded systems can be completely implemented in hardware with this approach. Nonetheless, the more complex the application is, greater is the probability it will need some sort of run-time system to adequately support its execution on top of a (soft-core) processor.

From the software perspective, this research group has been exploring methodologies and tools to sustain the development of embedded systems as aggregates of reusable components. One of the main challenges surmounted along this long-term research concerns architectural transparency for the run-time support system. In the realm of embedded systems, run-time support must often be provided under distinct architectures. For instance, an application running on a simple 8-bit microcontroller will probably be supported by a run-time library, while a multitask application running on a 32-bit microprocessor will probably require a microkernel. In order to enable application portability, a component-based embedded system must present the same interface in both cases. Indeed, the component should be the same, since its interface and behavior remains the same; only the software architecture is different. Our approach has so far enabled the development of run-time support systems whose architecture can be defined according to particular application needs.

Embedded applications usually don't find adequate run-time support on all-purpose operating systems and hardware. Application-Oriented System Design (AOSD) is an useful solution for this kind of applications, as demonstrated in previous works [3] [4] [8] [2]. Application-Oriented Operating Systems (AOOS) are targeted towards the applications, which means they are composed only from selected software components that are adapted to finely fulfill application requirements. Additionally, an AOOS may be supported by a hardware platform created by the composition and synthesis of proper hardware components. Delivering each application a tailored run-time support system, besides requiring a comprehensive set of well-designed software and hardware components, also calls for a sophisticated tool that will help the developer in creating component configurations, a task that includes the identification, selection, configuration, adaptation, and composition of

those components.

In this work we present an approach to the configuration management of component-based embedded systems which enables partial automation of the development process of mobile and embedded applications, contributing to an increase in productivity. Also, the ability to easily generate optimized versions of an operating system and a hardware platform for each of the applications that are going to use is of great value in the domain of high performance embedded computing, since it results in performance gains and resource usage optimization.

This paper is organized as follows: Section 2 briefly describes AOSD, our approach to the design of component-based embedded systems. Section 3 presents the main concepts and structure of a tool that is able to configure and generate embedded systems. Section 4 describes the details of a prototype implementation. Section 5 presents a case study that demonstrates the use of this tool and the last section presents some conclusions and a road map for future work.

2 AOSD

The Application-Oriented Systems Design method (AOSD) proposes strategies to define components that represent significant entities in different domains. By applying variability analysis, as defined in the Family-Based Design (FBD) [7], AOSD allows the modeling of independent abstractions and organizes them as family members. Even independent and separated, these abstractions may have dependencies from the environment to which they are applied. But abstractions that have environment dependencies will be reused with difficulty in different scenarios. To reduce environment dependencies and to increase re-usability of abstractions, AOSD adds to the decomposition process the main concern of Aspect-Oriented Programming (AOP): aspects separation. With this, it is possible to identify scenario variations and to model them not as family members, but as scenario aspects.

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. One of the first strategies is the one proposed by [3], the EPOS (Embedded Parallel Operating System). EPOS is a framework conceived through AOSD that combines concerns of FBD, OAP, Object Oriented Design (OOD) and Static Meta Programming (SMP) to guide the development of scenario independent component families that, by applying scenario adapters, can be used in different environments and provide architecture transparency [4]. Besides operating system components, it has been extended to deal with hardware [8], allowing 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. Indeed, with all these features, it seems a promising approach to help solving the problems that currently limit efficiency in SoC

development.

The main ideas behind AOSD are described below:

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.

Scenario adapters: as explained earlier in this article, Application-Oriented System Design 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 factored aspects to abstractions in a transparent 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.

Scenario adapters: as explained earlier in this article, Application-Oriented System Design 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 factored aspects to abstractions in a transparent 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. It allows application programmers to write their applications based on well-known, 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 realization that implements the requested interface subset at compile-time.

3 A CONCEPTUAL TOOL

Operating systems and hardware support designed according to the premises of Application-Oriented System Design, besides all the benefits claimed by software component engineering, have the additional advantage of being suitable for automatic configuration and generation. The concept of inflated interface enables an

application-oriented operating system and its hardware support 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. Our approach relies on a static configuration mechanism that allows the generation of optimized versions of the operating system for each of the applications that are going to use it.

The following is a detailed list of functionalities we have identified as requirements for a tool that addresses the problem: (i) **Requirements analysis** (automatic): applications must be inspected and their requirements regarding the support system should be extracted. (ii) **Component suggestions** (automatic): components that satisfy application needs should be identified. (iii) **Component selection** (semi-automatic): from a set of satisfactory components for an application, the most adequate ones should be automatically selected, and the developer must be allowed to select additional ones. (iv) **Component configuration and composition** (semi-automatic): default and critical components should be automatically selected, traits and features of components should be configured by the tool with default values and be modifiable. Also, the compliance to composition rules, such as exclusivity of some component members, should be maintained. (v) **Generation of systems and hardware** (automatic): based on configurations created by developers and components retrieved from a repository, system instances should be generated and hardware should be synthesized. (vi) **Cost estimation** (automatic): estimations of component costs, such as energy and silicon area, should be provided to the developer to assist him in choosing between components or between component members that achieve the same task. (vii) **Configuration validation** (semi-automatic): application and component dependencies should be tracked and presented to the developers, as well as other configuration problems and restrictions, such as restrictions regarding the chosen target hardware platform. The dependencies that can be automatically solved should be, while the others should be left to the developer to deal with.

And three main non-functional requirements to be considered: (i) **Specific Graphical User Interface** (GUI): one that has good usability and leads to efficiency. (ii) **Feedback to the developer**: the developers should know what is happening and, therefore, the actions that the tool automatically takes should be notified to them. (iii) **Automation**: some steps should be done automatically or by default.

We have organized these functionalities in three conceptual modules: `Analyzer`, `Configurator`, and `Generator`. Figure 1 illustrates how these modules could be used to assist an automatic or semi-automatic generation procedure of a system instance to support the runtime of a specific application.

An application's source code, previously written based on the interfaces of components from a repository (the OS API), can be submitted to the `Analyzer` module. This module searches for references to the interfaces, identifying what features are necessary from each component, and elaborates a requirement specification that

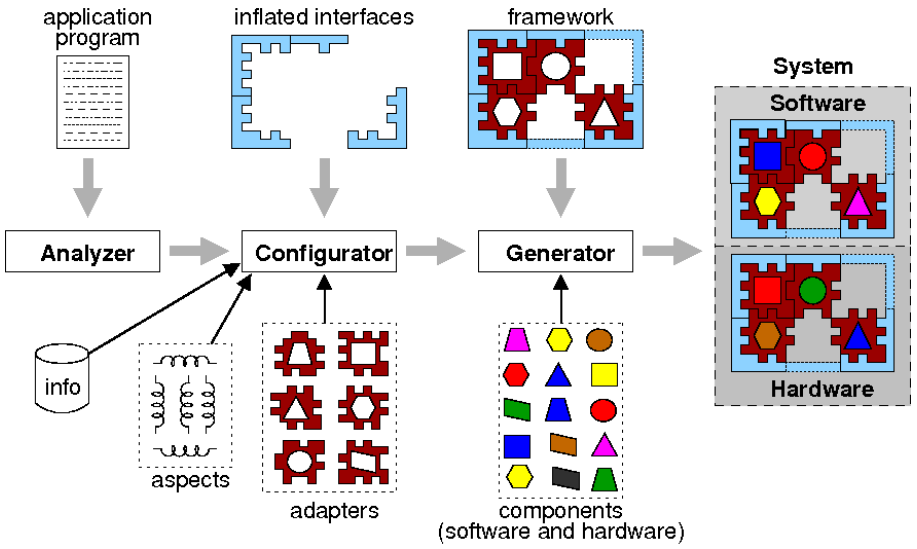


Figure 1: An overview of the tool

includes methods, types, and constants used by the application. If a requirement may be satisfied only by one component from the repository, the `Analyzer` can automatically choose it. If there are more than one, the components that meet the requirements, along with cost estimations, are presented to the developer so that he can choose the most adequate one. The primary output of the `Analyzer` is a set of application dependencies, since the application depends on some components in order to work properly. These dependencies feed the `Configurator`.

The `Configurator` module is logically divided in a `Validator` and a `Configuration`. The components chosen with the help of the `Analyzer` are added to the `Configuration` and used by the `Validator` to build a dependency tree. By doing this, the `Validator` is able to keep track of (1) aplication dependencies, (2) component dependencies, and (3) component composition rules. (1) refers to the dependencies that exist between application requirements and system components. (2) refers to the dependencies that may exist between components and their traits and features. (3) refers to the rules that should be followed when creating the `Configuration`, since some component members may be exclusive, and critical components cannot be touched. Some components are critical to any configuration, while others are critical to specific target platforms.

Although the `Validator` leads the developer to choose only the components that are necessary and adequate, he can interact with the `Configurator` to add and remove components from the `Configuration` and modify their features and traits.

The last step in the system development process is accomplished by the `Generator` module. Internally, it works by generating a set of keys that represent the current valid `Configuration`, binding the interface used by the application to specific component members existent in the repository, and activating the scenario aspects eventually identified as necessary to satisfy the constraints dictated by the application

or by the configured execution scenario. On the hardware side, the `Generator` produces a list of mediators that were included in the `Configuration`, specifying which ones are associated to IP blocks. These blocks are reusable hardware components, specified in a Hardware Description Language (HDL) or in an intermediate format, such as netlists.

The last step in the system development process is accomplished by the `Generator` module. Internally, it works by generating a set of keys that represent the current valid `Configuration`, binding the interface used by the application to specific component members existent in the repository, and activating the scenario aspects eventually identified as necessary to satisfy the constraints dictated by the application or by the configured execution scenario. On the hardware side, the `Generator` produces a list of mediators that were included in the `Configuration`, specifying which ones are associated to IP blocks, which are simple reusable hardware components, specified in a Hardware Description Language (HDL) or in an intermediate format, such as netlists.

The `Generator` module translates the keys into parameters for a statically metaprogrammed component framework and executes the compilation of a tailored system instance. In addition, whenever a SoC needs to be tailored, the `Generator` produces a synthesis configuration file that holds the parameters for configuring IP blocks and the information necessary to glue the IPs in a SoC. Written in a hardware description language, this configuration file and the selected IP blocks are handed over to a third-party tool, which performs the translation of the SoC specification into *netlists*. Finally, by considering the target PLD technology, these netlists are translated into a configuration *bitstream*.

4 PROTOTYPE IMPLEMENTATION

A proof-of-concept Java implementation of the tool described in the previous section has been developed and some details of its operation at the current development stage are given here.

Analyzer

The `Analyzer` 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 that refer to methods and/or constants from the 'System' namespace. By doing this, it is able to identify the usage of the operating system API. The references to the API are extracted by an external Perl script and output in the form of a XML file. The file is used by the tool to generate application dependencies, and the subsequent suggestions of satisfactory components.

Figure 2 depicts the `Analyzer` GUI. This interface allows a developer to specify application implementation files and include directories containing header files, to

visualize the code (right side), and to ask for requirements extraction. The requirements are exhibited in a tree structure (left side), which shows component families and members that satisfy the requirements. From the tree, the user may indicate the families and members to be included in the configuration.

We can observe a message console at the bottom of the figure. This console also accompanies the `Configurator` and `Generator` interfaces and shows all the relevant notifications, warnings, and error messages that must be delivered to the developer, relieving him from having to deal with dialog box windows.

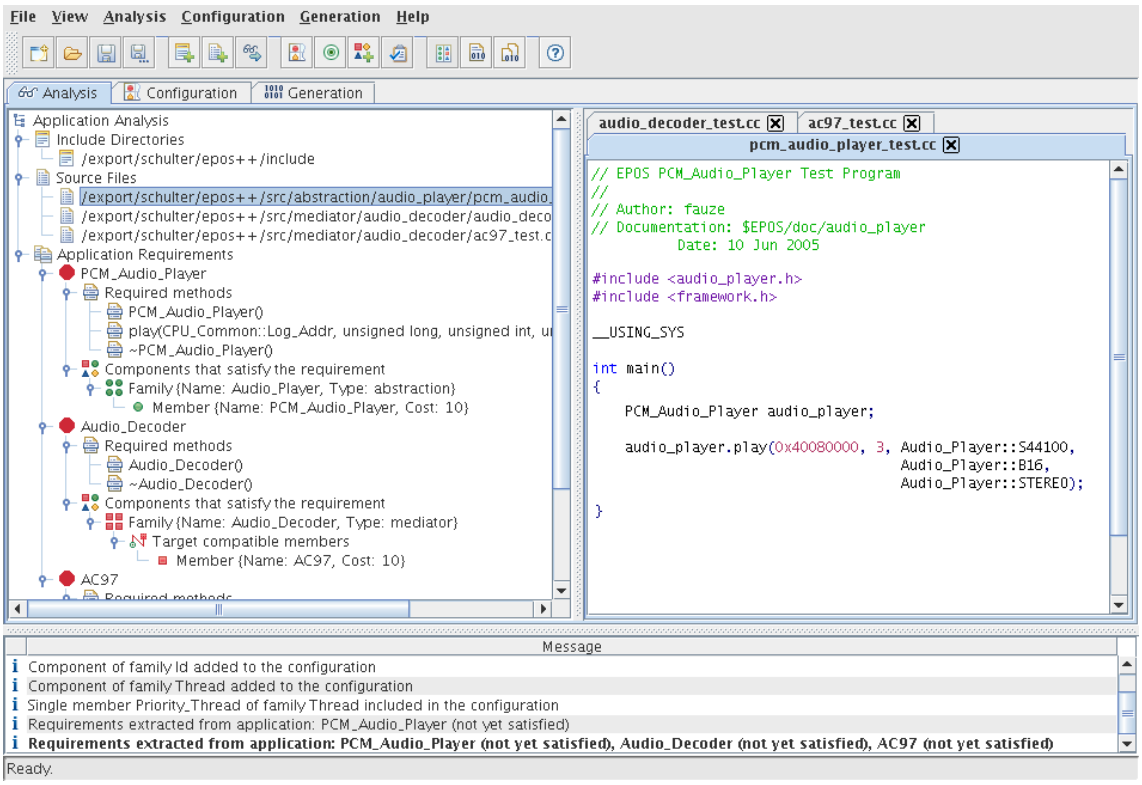
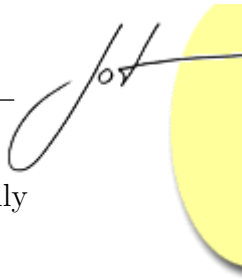


Figure 2: The `Analyzer` interface

Configurator

The strategy used to describe components in a repository and their dependencies plays a key role in making the configuration process possible. The description must be complete enough so that the `Configurator` will be able to automatically identify which abstractions better satisfy the requirements of the application without generating conflicts or invalid configurations and compositions.

The component description language we adopted is based on XML and is focused on describing each component individually. As shown below, a component is defined by a `family` and its set of `member`. A family description also includes an `interface` declaration, an optional set of `dependency`, an optional set of `trait`, and a `common`



package that holds `type` and `constant` declarations that are common to all family members.

```
<!ELEMENT family (interface, dependency*, trait*, common, member+)>
<!ELEMENT interface (type, constant, constructor, method)*>
<!ELEMENT common (type, constant)*>
<!ELEMENT member (super, interface, trait, cost, feature, dependency)*>
```

A family's interface summarizes the features of the whole family, including `constants`, `constructors`, and `methods`. Each member of a family is also described by an `interface`, which designates a total or partial realization of the family's interface. Additionally, a member is described by an optional `super` declaration, an optional set of `trait`, `dependency`, and `feature` declarations, and a required `cost` declaration.

The `super` element determines the inheritance from other members in the family, while a `trait` designates a configurable information that can be set by the developers, via the `Configurator`, in order to influence the instantiation of a component. A trait can also be used to specify configuration parameters that cannot be automatically figured out, such as the number of processors in a target machine or the amount of memory available.

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.

If the family is hardware mediator type or hardware IP type, the member descriptions must also declare `arch` and `mach`, which specify the architecture and machine the member is compatible with. More details and examples of the description language can be found in [10] [11].

The `Configurator` GUI is depicted in Figure 3. It allows the developer to see the list of components included in the configuration (left side). Through the configuration desktop pane (right side), all the components, as well as the configuration's target platform, can be viewed and modified with the help of their respective configuration frames, although the default values for their features and traits are often adequate. The `Configurator` automatically includes the minimum necessary components and allows the developer to include other ones. However, currently it does not identify unnecessary or redundant components. Despite not being implemented, this could be achieved simply by analyzing the dependencies.

The `Validator` also has a graphical interface that opens up in the `Configurator` window. It shows the current collection of dependency problems and other miscellaneous ones, such as incorrect information describing component, target, or general configuration characteristics.

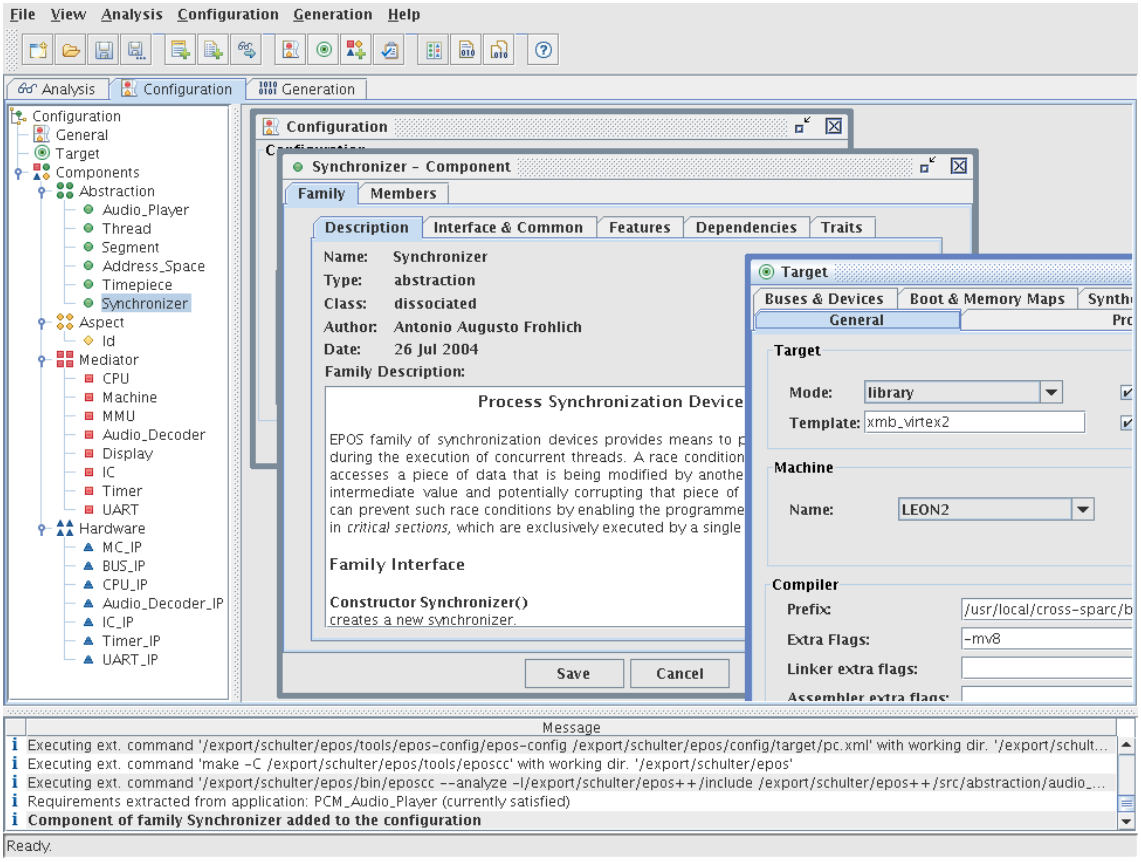


Figure 3: The `Configurator` interface

Generator

The `Generator` allows a developer 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. Before these generation processes can be executed, a set of directive files are created, including traits files and a 'keys file' that guide the generation. 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. A limitation of this `Generator` is its inability to estimate properties (memory and silicon area, latency, power consumption, etc.) of the final system before really building it.

The `Generator` interface, as depicted in Figure 4, is very straightforward. It shows informative output from the processes launched by their respective buttons.

Some logic is applied to allow the actions a developer can take. For instance, it is not permitted to build the system or hardware before the 'Config' action is taken, since that action causes the generation of the directives needed by the building processes.

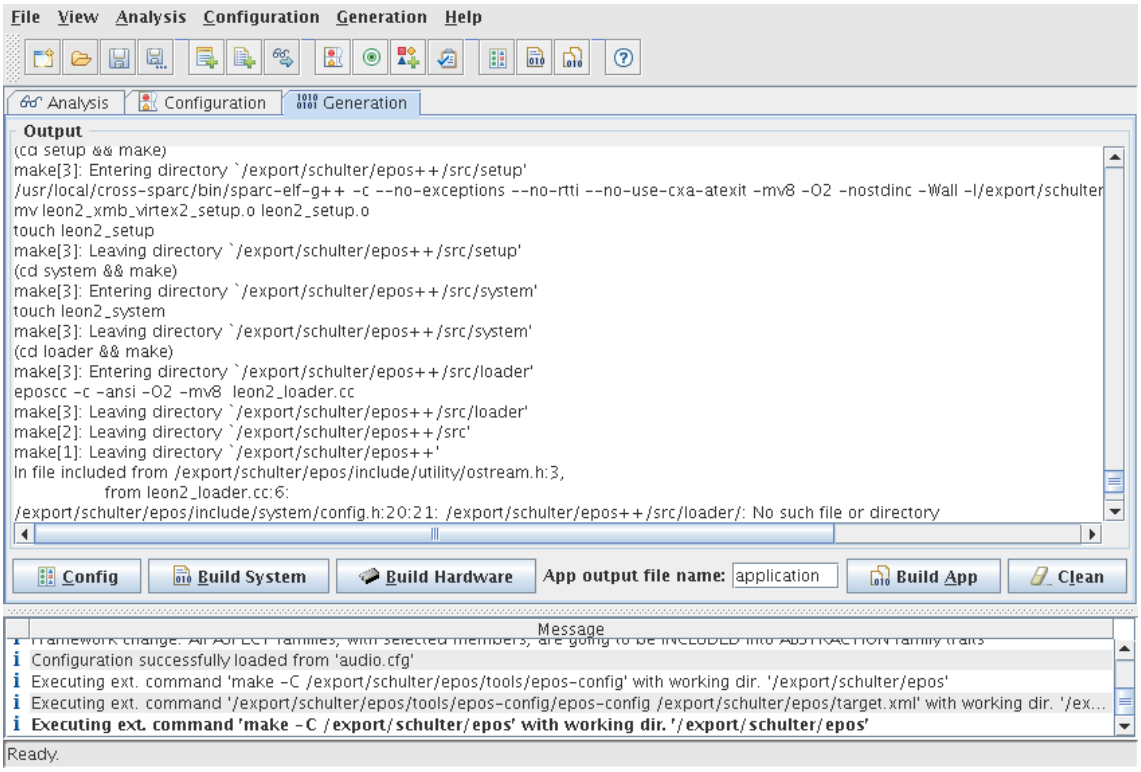


Figure 4: The `Generator` interface

This tool was designed to assist developers in executing the procedure depicted in Figure 1 in a simple, direct, and semi-automatic manner. The GUI and its usability have an important role, since it must greatly reduce the work a developer is required to do in comparison to what would be performed manually without tool assistance. Several conveniences, such as action keys, pop-up menus, icons, and intuitive responses are provided by it, and all the elements are organized in a way to make a clear separation of the activities related to analysis, configuration, and generation steps, although the developer is able to go back and forth through the steps.

5 CASE STUDY AND RESULTS

In order to evaluate the prototype implementation described in Section 4 and its support for the requirements listed in Section 3, we performed some case studies involving the automatic or semi-automatic generation of different kinds of embedded applications, such as multimedia, automotive control, wireless sensor networks, energy aware mobile systems [1] [5] for different platforms/architectures, such as

AVR, MIPS, SparcV8, PPC405, and IA32.

In this paper we present a simple audio player application. The case study we will describe here was performed with the help of the EPOS component-based operating system and a Xilinx prototyping board as the target hardware platform. In order to avoid similar figures, all screenshots previously shown in this paper depict this case study.

The Xilinx board used as target contains a Virtex-II FPGA. Besides two built-in PowerPC 405 CPUs in that board (not used), a LEON2 soft-core processor was synthesized along with other IP blocks to create a system-on-a-chip. This board is targeted towards multimedia application prototyping and some of its available devices are: Memory Management Unit (MMU), Floating Point Unit (FPU), Time Stamp Counter (TSC), Interrupt Controller (IC), Data Service Unit (DSU), Watchdog, Universal Asynchronous Receiver/Transmitter (UART), Timer, Network Interface Card (NIC), Audio Decoder, Video Decoder, and Display Controller [6].

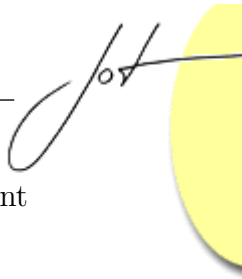
The application's source code is written in C++ and is listed below (the same that appears in Figure 2). As we can see, only one method from the EPOS API is called, the `play(Log_Addr sound, Seconds length, Sample_Rate sample_rate, Bit_Depth bit_depth, Sound_Mode mode)` method from the `PCM_Audio_Player` abstraction. The parameters indicate that a `16-bit stereo` sound stream sampled at `44.1 khz` and stored at the `0x40080000` address should be played for `3` seconds.

```
// PCM_Audio_Player Test Program

#include <utility/ostream.h>
#include <audio_player.h>
#include <framework.h>

__USING_SYS

int main()
{
    PCM_Audio_Player audio_player;
    audio_player.play(0x40080000, 3, Audio_Player::S44100, Audio_Player::B16, Audio_Player::STEREO);
    return 0;
}
```

The `PCM_Audio_Player` abstraction provides functionalities related to controlling the reproduction of uncompressed digital audio streams represented with Pulse Code Modulation (PCM). PCM is a standard for digital audio in computers and compact disks (CD) and is the usual output of audio decompressors, such as MPEG-1 Layer 3 (MP3) decoders.

The following is a description of the actions and interactions between a developer and the tool when the audio player application is submitted to begin the procedure:

- The `Analyzer` identifies a reference to the `play()` method and suggests the `PCM_Audio_Player` abstraction component to satisfy this requirement. (see Figure 2)



- The developer interacts with the `Analyzer` and confirms that this component can be added to the `Configuration`.
- The `Configurator` acknowledges that the `PCM_Audio_Player` component is now an application dependency and adds it to the `Configuration`.
- The developer specifies that the target platform is a Virtex II Pro. (see Figure 3)
- The `Validator` informs the developer that `PCM_Audio_Player` depends on the `Audio_Decoder` mediator component.
- Among the `Audio_Decoder` members, the `Configurator` suggests and automatically selects the `AC97` member, because it is the only mediator compatible with the target's architecture and available devices. The developer opts for maintaining the selection of this type of decoder.
- The `AC97` member from the `Audio_Decoder` component depends on the `AC97` member from the `Audio_Decoder_IP` hardware component.
- The developer authorizes the inclusion of the `Audio_Decoder_IP` component and its `AC97` member is automatically selected.
- Besides these player and decoder components, three critical components are automatically included by the `Configurator` and cannot be removed: `CPU`, `Machine`, and `Thread`. The `CPU` component is the most critical. The `Machine` component mediates the platform architecture and is automatically configured with trait values (architecture, processor, memory, etc.) that conform to the target the developer has previously chosen. The `CPU` and `Machine` are hardware mediators and, because of this, need to be compatible with the target's architecture. The `Configurator` automatically selects their members based on this requisite. Finally, the `Thread` component is included to support the runtime of the application, even though it is not multi-threaded.
- When the developer finishes interacting with the `Configurator` to solve the dependencies that have appeared, the `Validator` allows the generation process to occur.
- The developer pushes some of the `Generator` buttons, causing the system generation. This consists of compiling the highly customized instance of EPOS, calling the *Xilinx Synthesis Tool* (XST) to build the hardware, compiling the application, and linking the resulting object files to create a bootable image. This image is a few kilo bytes in size and is ready to be uploaded to the hardware's memory and tested (see Figure 4).

Figure 5 depicts the components included in this case study's configuration and the dependencies existent between them.

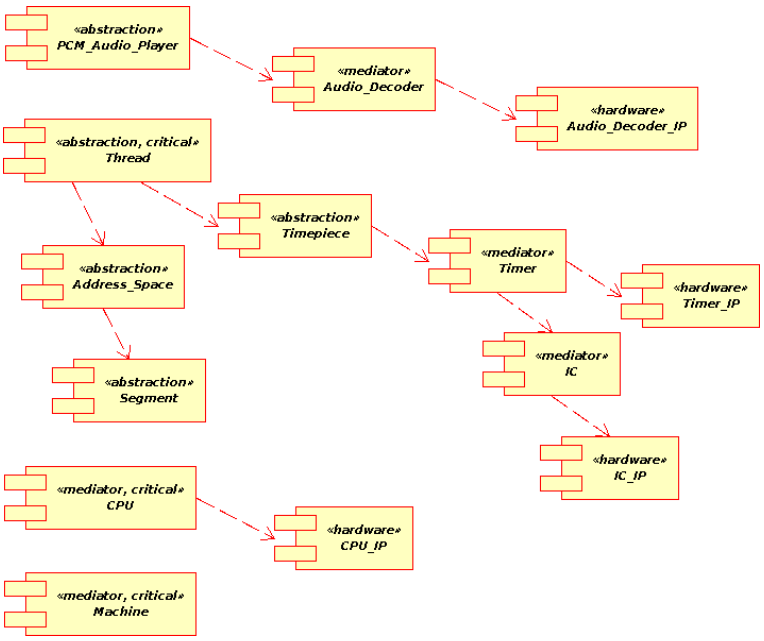


Figure 5: Component diagram of the audio player system configuration

We found that this tool is adequate for the use case described here, since it satisfies the requirements elicited in Section 3: application analysis; component suggestions, component selection, configuration, and composition; generation of system software and hardware; configuration validation; specific graphical interface; feedback to the developer; and partial automation. Cost estimations, though, is a requirement that currently is not fully supported, since no meaningful estimations for each of the components are calculated.

Several other applications, components, and target platforms were tested with the tool. Its strong points are its fast and simple operation and its internal design that features good maintainability and extensibility. Another aspect to consider is the possibility of using it to configure and generate other component-based operating systems. This would be possible if such systems provide some of the facilities EPOS does: a component repository properly described in a language the tool knows how to interpret and a set of makefiles that provide building rules.

6 CONCLUSIONS

In this paper we dealt with the problem of developing embedded systems. Even with component-based design, current solutions do not address efficiently architectural transparency, performance, configuration and generation of application oriented embedded systems. We have shown the basic concepts of Application-Oriented System Design methodology that was developed to solve these problems and we focused here on the final phase of development.

We have presented a tool that assists developers in configuring and generating software and hardware support for embedded systems taking as base a collection of reusable components developed according with the Application-Oriented System Design methodology, their dependencies and composition rules. The prototype implements the requirements listed in Section 3, and effectively identifies, selects, configures, adapts, and composes those components, generating real and functional embedded systems. Steps are performed automatically whenever possible by the current prototype and some hints are given when design decisions are required. The developer can focus on his application, not on the underlying support system, which leads to a faster development process and, incidentally, shorter time-to-market for commercial products.

As a case study, an example of a configuration process for a simple but real application was presented, as well as the current limitations of the model due to the need of manual selection when more than one different component satisfy the required interface. We believe that it represents an important step in the direction of a fully automated generation since we have promoted the generation of software and hardware support and SoC instances in the same generation flow according to a well defined methodology. The strategy currently realized does not represent a complete solution to automate the process of generating SoC-based embedded systems (whereas the programmer still takes some decisions regarding the selection of more than one different realization for a component). We believe a more detailed costs model for components and the inclusion of a smarter algorithm to perform design exploration could automatically solve most of the situations. However, some decisions will seldom be fully automatic, such as choosing the best real-time scheduler component that garantees deadlines will be meet based only on application source code and on component interfaces.

The components existent in our repository were identified and defined with the AOSD approach, which starts with domain analysis, identification of families, identification of members and aspects. Since great attention is required when performing these steps, the actual specification of components and their features does not require much effort. A developer needs only to implement their code and add some lines of description in configuration rules files. The maintainance of these components is also not a hard task, since abstraction components will seldom be modified, and hardware mediator ones have different implementations for each platform they refer to.

Our road map for future work includes simple improvements, such as the implementation of a convenient GUI that shows numbers and charts that better represent the system, to make the `Configurator` responsible for finding out if there are any unnecessary, redundant, or not very adequate components, to include convenient repository manipulation functionalities to help developer in managing component implementations, and to provide a simple source code editor for the `Analyzer`.

Finally, the results obtained with Application-Oriented System Design, the EPOS framework, and the tool described in this paper are so far encouraging. Research

in progress is looking forward to improve the costs model for components and allow intelligent design space exploration for automatic and adaptive component selection and partitioning. With this, our approach would not only automatically generate systems that match applications requirements, but also design requirements, such as as maximum silicon area or power consumption.

REFERENCES

[1] Lucas Francisco Wanner Arliones Stevert Hoeller Junior and Antônio Augusto Fröhlich. A hierarchical approach for power management on mobile embedded systems. In *Proceedings of the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 265–274, Braga, Portugal, 2006.

[2] Roberto de Matos Danillo Santos, Rafael Cancian and Antônio Augusto Fröhlich. Advantages and disadvantages of application-oriented system design in embedded systems design. In *Proceedings of 4th International IEEE Conference on Industrial Informatics*, pages 904 – 909, Cingapura, 2006.

[3] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. PhD thesis, GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Germany, 2001.

[4] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario adapters: Efficiently adapting components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA, 2000.

[5] Augusto Born de Oliveira Lucas Francisco Wanner, Arliones Stevert Hoeller Junior and Antônio Augusto Fröhlich. Operating system support for data acquisition in wireless sensor networks. In *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 582–585, Prague, Czech Republic, 2006.

[6] T. Tierens P. Pelgrims and D. Driessens. Overview: Excalibur, leon, microblaze, nios, openrisc, virtexii pro. Technical report, DE NAYER Instituut, 2003.

[7] David Lorge Parnas. On the design and development of pro- gram families. In *IEEE Transactions on Software Engineering*, pages 1–9, 1976.

[8] 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, Aizu,Japan, Aug. 2004.

[9] P. Pop. Embedded systems design: Optimization challenges. CP-AI-OR 2005 Invited Talk, May, 31 2005. Embeded Systems Lab (ESLAB), Linköping University, Sweden.

[10] Gustavo Fortes Tondello and Antônio Augusto Fröhlich. Configuration management of embedded operating systems using application-oriented system design. In *Proceedings of the 5th Argentine Symposium on Computing Technology (part of the 33rd Argentine Conference on Computer Science and Operational Research)*, Córdoba, Argentine, 2004.

[11] Gustavo Fortes Tondello and Antônio Augusto Fröhlich. On the automatic configuration of application-oriented operating systems. In *Proceedings of the 3rd ACS/IEEE International Conference on Computer Systems and Applications*, pages 120 – 123, Cairo, Egypt, 2005.

ABOUT THE AUTHORS

Alexandre Schulter works as an IT analyst at Dataprev, a brazilian public company. In the past he has worked as a researcher and software developer at several laboratories in the Technological Centre, Federal University of Santa Catarina, Brazil. He holds the MSc (2006) and Bachelor (2003) degrees in Computer Sciences, and his areas of interest include Information Systems, Component-based Systems, Computer-Aided Software Engineering, Distributed Systems, Grid Computing, and Security. schulter@inf.ufsc.br. See also http://www.inf.ufsc.br/~schulter.

Rafael Luiz Cancian is a PhD canditate in Electrical Engineering at the Federal University of Santa Catarina, Master (2000) and Bachelor (1997) in Computer Sciences at the Federal University of Santa Catarina (UFSC). Currently he is a professor of the Computer Sciences Department at University of Vale do Itajaí - UNIVALI (Itajaí, Brazil), and associated researcher of the Laboratoty of Software and Hardware Integration (LISHA/UFSC) and the Laboratory of Embedded and Distributed Systems (LSED/UNIVALI). cancian@das.ufsc.br. See also http://www.das.ufsc.br/~cancian.

Marcelo Ricardo Stemmer is a PhD in Industrial Automation (1991) (WZL / RWTH-Aachen, Germany), Master (1985) and Bachelor (1982) in Electrical Engineering (1985) (Federal University of Santa Catarina - UFSC). Currently he is a professor at the Department of Automation and Systems (DAS) of the Federal University of Santa Catarina (Florianópolis, Brazil), and head of the Intelligent Industrial Systems (S2i) research group at UFSC. marcelo@das.ufsc.br. See also http://www.das.ufsc.br/~marcelo.

Antônio Augusto Medeiros Fröhlich is a PhD in Computer Engineering (Technical University of Berlin), MSc in Computer Science (Federal University of Santa Catarina), and Bachelor in Computer Science (Federal University of Rio Grande do Sul). Currently he is an associate professor at the Computer Science Department of the Federal University of Santa Catarina (Florianópolis, Brazil), head of the Laboratory for Software/Hardware Integration (LISHA) at Federal University of Santa Catarina, and external research associate at the Fraunhofer FIRST within the Software Engineering Group (Berlin, Germany).
guto@lisha.ufsc.br. See also http://www.lisha.ufsc.br/~guto.