EPOS Repository Structure

Rafael Luiz Cancian, Marcelo Ricardo Stemmer
Departament of Automation and Systems (DAS)
Federal University of Santa Catarina (UFSC)
Florianópolis, SC 88000-000
Email: {cancian, marcelo}@das.ufsc.br

Antônio Augusto Medeiros Fröhlich
Department of Informatics and Statistics (INE)
Federal University of Santa Catarina (UFSC)
Florianópolis, SC 88000-000
Email: guto@lisha.ufsc.br

***Resumo*—The increasing complexity of embedded systems demands the use of adequate design methodologies and tools to support them. The component based design and the design space exploration are central points in the development of embedded systems, and are closely related to the modeling of the component repository. In this paper we present three main advances in the EPOS (Embedded and Parallel Operating System) repository to support design space exploration: the creation of the repository from a domain model, a more complete representation of software, synthetizable and physical hardware components, architectures and programmable logical devices, and keeping a history of designed systems and estimated component costs.**

***Index Terms*—Embedded systems, Component repository, Design space exploration**

I. Introduction

Embedded systems are equipments that execute a dedicated application. The evolution of the technology and the demand for new applications have led to a constant increase in the complexity of embedded applications. The component based development has been a widely accepted and used approach with the objective to treat this complexity. In this approach, reusable software and hardware components form the infrastructure that will compose several applications, allowing to the developer to concentrate solely in the application logic. However, the amount of combinations in which components and their configurations can be composed to support a new application is usually huge. Therefore, the exploration of the design space is a central aspect in the project of embarked systems. The design space exploration aims at to find an optimal solution that simultaneously minimizes some costs of the system. The most common costs are the consumption of energy, memory area, execution time and latency. To perform this exploration, it is necessary to generate different possible solutions, composed by different components with different configurations, executing in different hardware platforms and then to evaluate each solution based on the interest costs. Based on these evaluations, a multi-objective optimization mechanism is used to find the best solution.

Unfortunately, the design of components and the design space exploration include several problematic points. Components are not reusable by themselves. They must to be designed to be reusable in some domain. Indeed, a technique as the domain engineering should precede and guide the development of components and serve as base to the component repository. In addition to the components themselves, the repository needs to include information that will be used by design exploration techniques. Repositories for embedded systems can contain software, synthetizable hardware (IPs), and physical hardware components. Each one of them have different characteristics and usually nor all information is available for all the components. The costs of the components also depend on the context they are to be used.Other costs normally are not considered for the design space exploration, since their values cannot automatically be obtained from compilation and synthesis tools. These costs include some characteristics of great importance to the designer of embedded systems, as the financial cost of the components, the physical dimension and the weight of the system.

Our research group has developed embedded systems using the Application Driven Embedded System Development (ADESD) methodology. To support this methodology, a computer aided design tool has been developed. However, until the last version of such tool before the development of this work, it did not treat the issues previously presented. The repository included only the essential information of software and IPs components and the developer had to to specify the target platform a priori, to manually choose between alternative components and to specify their configuration, in way that design space exploration simply did not exist. This work is developing a new tool to support previously presented issues, and all advances are based on a new structure and specification for the component repository, that is the focus of this paper.

The approach used in this work allows to create and to maintain a component repository that is based on more representative entities of a domain, since it is generated from a domain model. The more complete representation of software, synthetizable hardware and physical components allows to a more extensive design space exploration and also provides the infrastructure for some future developments. The representation of many estimates for each cost and the feedback from the designer allow to the adaptation of these costs, including design spaces that usually cannot be estimated by compilation, synthesis and simulation tools. A history of estimated costs is kept for each system design, as well as information of the design itself, allowing to the adaptation of the costs for new designs and the use of alternative approaches, as the use of the same hardware platform for designs of the same product family, as suggested by Platform Based Design (PBD) [1], or

the development of a specific and minimum platform for each design, as suggested by ADESD.

The basic concepts needed to understanding the ADESD methodology and the EPOS component repository are presented in the section II. The advances in EPOS repository to support design space exploration are presented in the section III and some final statements are presented in the section IV

II. ADESD e EPOS

The Application Driven System Design (ADESD) methodology [2] aims at to supply adequate support to specific domain applications. Application driven systems are composed exclusively by necessary software and hardware components that are tailored to match exactly the requirements of a specific application. ADESD uses strategies to define components that represent significant entities in different systems and execution scenarios, what begins with the Domain Engineering [3]. The variations in the domain are treated as defined in the Family Based Design [4]. Significant domain entities that are independent of the execution scenario can be shaped by the ADESD as *system abstractions* and then organized as members of component families, that have an *inflated interface* [2] for all its members.

Even with a criteriously separation, these abstractions can contain dependences to non-functional properties and to the execution scenario which they are applied to, i.e, architectural dependences. To reduce these dependences and to increase the reusability of components, ADESD aggregates three main approaches: (I) *separation of aspects*, as defined in the Aspect Opriented Programing (AOP), for the isolation of the non-functional properties; (II) *scenario adapters* [5], that configure and adapt components to a specific execution scenario, possibly by applying *aspects* or by activating *configurable features* [6]; and (III) *hardware mediators*, that encloses architectural dependences [7].

Briefly, systems designed using ADESD are composed by *software or hardware components* that are *members* of a *component family* and relies on a component repository. Execution scenario independent components are *system abstractions* and the ones that encapsulate specific architectural dependences are *hardware mediators*. Both can be configured by *scenario adapters*, that applies *aspects* or activates *configurable features* and are also configurable through specific *traits*. One *component family* appears to the developer as a super-component, therefore presents an *inflated interface* for all its members, delaying the decision on which component will be used by the application. The ADESD does not make any distinction between software elements and hardware architectural elements. In ADESD there only exist components that may have dependencies of other components.

One of the first practical strategies using AOSD is EPOS (Embedded Parallel Operating System). EPOS is a framework conceived through AOSD that combines several concerns to guide the development of scenario independent component families that can be used in different environments and provide architecture transparency [5]. In addition to operating system components, it has been extended to deal with hardware IPs [8], allowing to 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.

III. EPOS Repository for Design Space Exploration

Some recent advances in the ADESD supporting tool aim at the design space exploration in embedded systems, and many of them are based on changes in the representation of components and in the repository structure. In this paper we present three advances that had been carried through in the repository: (I) generation of the repository from a domain model, (II) more complete representation of software, IP and physical hardware components, architectures and programmable logical devices, and (III) keeping a history of designed systems and estimated component costs and adapting them.

The first advance deals with the generation of the repository from a domain model. Repositories of components are not static. They change always existing components are modified or components are included or removed. Maintaining component costs manually in the repository is tedious and susceptible to errors. Our approach begins with the creation of a domain model specified in UML. This model must be based on class diagrams and needs to define one package with one diagram for each components family and to include in this diagram all the members of that family. The tool identifies the components of the domain (based modeling standards that design must use) and generates the structure the repository, which includes: (I) a hierarchic structure of folders (the same hierarchy of families in the model); (II) files of components source-code and debuggers, with classes, (empty) methods, attributes, constants and file dependences; and (III) a hierarchic structure of XML files with metadata that describe the components, as described in the next paragraphs. Several UML tools perform reverse engineering, what allows to keep the domain model synchronized with the components. Currently, the repository of the EPOS includes around 50 software components, 20 IPs and 15 physical components. The interface of the components, their dependences and configurations can be automatically extracted from the domain model. However, other information, as price of licenses and documentation, for instance, do not exist in the domain model and need to be set manually. Information about physical components, as packaging, pins and electric features are also not in the domain model and need to be set manually from data sheets.

The second advance deals with a more complete representation of components. To allow to the automatic selection and configuration of components, the repository has to include the interface of the components, their dependences and theirs possible configurations, that is, traits and configurable features [6]. To allow to the design space exploration, a more complex structure modeling components was developed, as depicted in the model of the figure 1. General `component` includes

everything that is common, as dependences, associations over families and members, inflated interfaces, documentation and configuration. `Virtual components` can be specialized into `software components` or `synthetizable components`, and have files associated to them, as source-code, debuggers and models, and information about the actual version, state of implementation, validation, known bugs and ToDo lists, since nor all the virtual components in a the repository are always 100% implemented and validated. The `hardware interface` is the base interface for both IPs and physical components, and specifies the general characteristics of pins and buses. IPs specialize these information to represent possible configuration (e.g., vhdl *generics*), silicon area and compatibilities to on-chip-bus standards. `Physical components` specialize them to represent electric features, as minimum and maximum electric amperage and voltage, resistance, packaging, physical dimensions and weight. Two other specializations were modeled. `Architectures` can be synthetizable components (soft-cores) or physic components (e.g. microcontrollers). Information include clock frequency range, features of the instruction set and the integrated peripherals. Every software component must be mapped to an architecture component to be executed in. `Programmable logical devices` (PLD) are physical components specializations, and information include total silicon area and some special characteristics, as the number of included BRAM. Every synthetizable component must be mapped to a PLD component to be synthesized in. This allows to map software to a physical architecture connected to other physical components to model the physical hardware platform, or to map software to a soft-core architecture connected to other synthetizable components to form a synthetizable hardware platform. In this case, all synthetizable component in the platform is mapped to a physical PLD to form the physical hardware platform.

Figure 2 depicts a simplified representation of a dependence graph with the different abstraction layers that compose the EPOS repository, starting from a single application dependency to a semaphore component. In this figure, nodes are component in the repository and edges are dependences. Hatched edges represent mutually exclusive dependences, or implementation alternatives, so the design exploration technique must further select only one of them, aiming at to optimize the solution. Dependence graphs and mapping graphs are automatically extracted from the XML files of EPOS repository using XML parsers and the *jGraphT* package. These graphs are then used to make the allocation, mapping and scheduling of tasks [9], generating solutions that will be optimized.

The third and last advance treated in this paper is the representation of system designs and component costs estimates. The figure 3 depicts a simplyfied version of the model used for `characterization` of the component costs. The costs normally used for design space exploration depend on diverse factors, that are others `attributes` of components. For example: the cost of memory usage of a software component depends on the instruction set of the architecture on which it

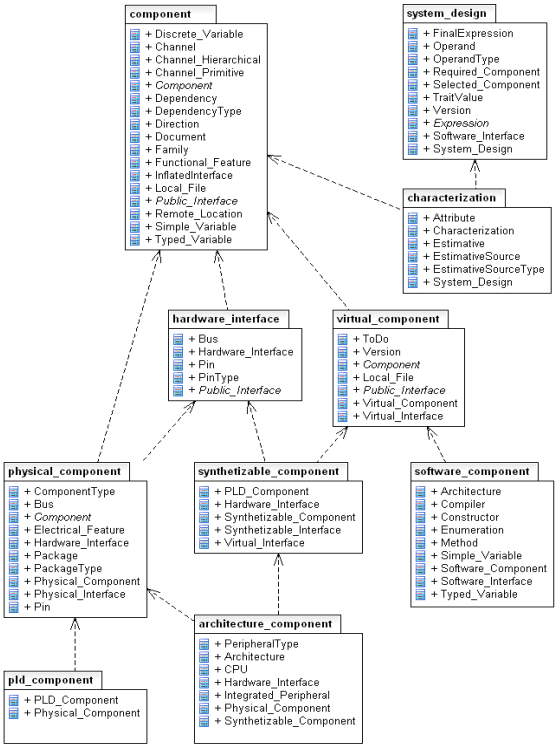

Figura 1: *Component repository packages*

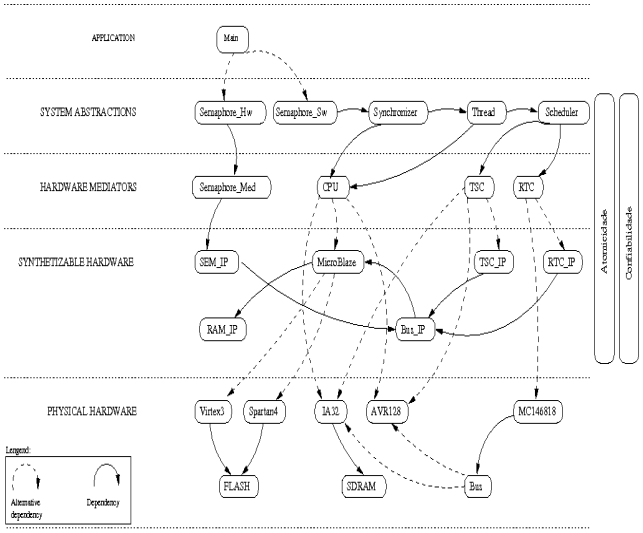

Figura 2: *Partial dependency graph representation*

is mapped to, among others factors. Therefore, for each cost there may exist several different `estimatives` of its value, depending on the value (level) of the factors that influence it in the context of one specific `system design`. Each estimate can be obtained from different `estimative sources`, as analytical simulators, RTL co-simulators or event the feedback from the designer. Each possible estimate source has a different precision. Therefore, there are better estimates than others. The component costs estimates are used by the tool to

infer the total cost of the embedded system being designed. For this, in addition to the cost numerical value, the repository stores a mathematical expression that represents it, defined in terms of its factors. This expression is adapted by the multi-objective mechanism used for design space exploration, that is a multi-objective evolutionary algorithm (MOEA). The history of known estimates is used as initial population to optimize new designs.

The basic characteristics of each system designed are stored in the repository. A `System Design` is used as context for the cost estimates but also for other purposes. The characteristics stored include its inputs and outputs. The inputs of a system design are: (I) the application dependences that correspond to inflated interfaces of component families; (II) resources restrictions specified by the developer, such as limits for execution time and memory; and (III) a list of aspects that must be activated, since nor all the non-functional properties can be automatically inferred from the interfaces invoked by the application. The outputs of a system design are the list of selected components, their configurations, dependences, mapping and scheduling. Since the hardware platform is of particular interest, the components that form this platform are stored separately for easy access. Finally, each design can be related to other designs. With this, the repository allows to representing the design of product families, or evolutions of the design of a product. Related designs can share the same hardware platform (as suggested by Platform Based Design) or can use the selected components of a previous solution optimized for a similar design as part of the initial population of solutions that will be optimized for the current design, possibilly increasing the convergence of the multi-objective evolutionary algorithm.

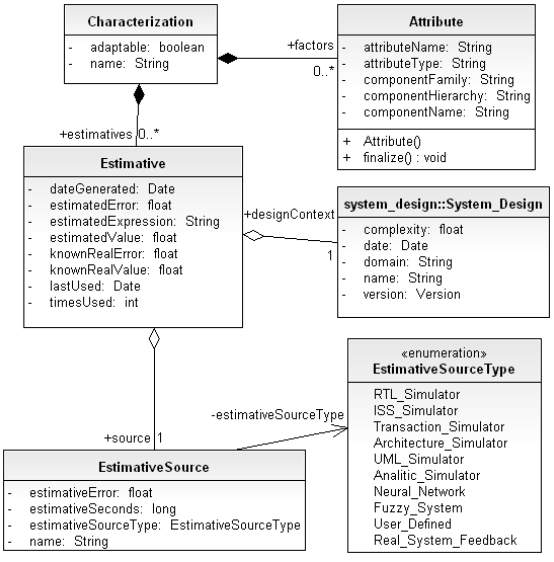

Figura 3: *Objetive-function characterization*

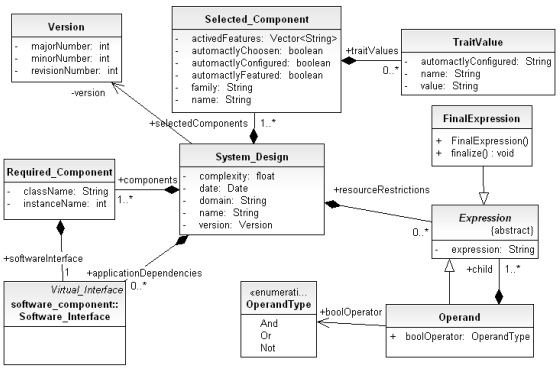

Figura 4: *System design*

IV. CONCLUSIONS

The contributions of this work include the creation and maintenance of the component repository based on domain models, which allows to components to correspond to more representative entities and therefore to be more reusable. The more complete representation of components, from high level software components to low level physical hardware components allows to a more extensive exploration design spaces and form the infrastructure for several future developments. Finally, the representation of adaptive costs and system designs and keeping a history of such representations allow other design spaces to be explored, since objective-functions are defined and adapted from the developer feedback and that different design approaches can be compared.

REFERÊNCIAS

[1] A. Ferrari and A. Sangiovanni-Vincetelli, "System design: Traditional concepts and new paradigms," in *Proceedings of the International Conference on Computer Design*, 1999, pp. 2–12.

[2] A. A. M. Fröhlich, "Application-oriented operating systems," Ph.D. dissertation, Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001, 200 pages.

[3] R. Pietro-Diaz, "Domain analysis - an introdution," ACM SigSoft Engineering Notes, Tech. Rep., 1990.

[4] D. L. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1–9, mar 1976.

[5] 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, USA, July 2000. [Online]. Available: http://www.lisha.ufsc.br/~guto/publications/sci2000.pdf

[6] 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, 2005, pp. 120–123.

[7] 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*, ser. Lecture Notes in Computer Science, vol. 3207. Aizu, Japan: Springer, Aug 2004, pp. 271–280. [Online]. Available: http://www.lisha.ufsc.br/~guto/publications/euc2004a.pdf

[8] ——, "On the automatic generation of soc-based embedded systems," in *10th IEEE International Conference on Emerging Technologies and Factory Automation*, Catania, Italy, September 2005. [Online]. Available: http://www.lisha.ufsc.br/~guto/publications/etfa2005a.pdf

[9] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.