Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 425401, 15 pages
doi:10.1155/2011/425401

Research Article

AADL Extension to Model Classical FPGA and FPGA Embedded within a SoC

Dominique Blouin,[1] Daniel Chillet,[2] Eric Senn,[1] Sébastien Bilavarn,[3] Robin Bonamy,[2] and Christian Samoyeau[4]

[1] *Lab-STICC/CNRS UMR3192, Université de Bretagne-Sud, Centre de recherche, BP 92116, 56321 Lorient Cedex, France*
[2] *Cairn Inria/Irisa, Université de Rennes 1, ENSSAT, 6 rue de Kerampont, BP 80518, 22305 Lannion, France*
[3] *Leat/CNRS UMR6071, Université de Nice-Sophia Antipolis, 250 rue Albert Einstein, Bt. 4, 06560 Valbonne, France*
[4] *InPixal, Immeuble "Le Germanium", 80 avenue des Buttes de Cosmes, 35700 Rennes, France*

Correspondence should be addressed to Daniel Chillet, daniel.chillet@irisa.fr

Received 26 November 2010; Revised 23 March 2011; Accepted 25 May 2011

Academic Editor: Koen L. M. Bertels

With the evolution of technology, the system complexity increased and the application fields of the embedded system expanded. Current applications need a high degree of performance, flexibility, and efficient development environments. Today, reconfigurable logic allows to meet the on-chip processing requirements with new benefits resulting from partial and dynamic reconfiguration. But the dimension introduced in the design of these systems requires more abstraction to manage their complexity and efficient models to provide reliable preliminary estimations. While classical multiprocessor systems can be modeled without difficulty, the use of partial run-time reconfiguration in heterogeneous flexible system-on-chips is generally not covered. The contribution of this paper is to address this with an extension of the AADL language able to model the reconfigurable logic, possibly considering dynamic reconfiguration and power consumption requirements. The proposed AADL model is divided into three levels to provide a generic and hierarchical approach separating the static and dynamic parts of current FPGAs. These levels are exposed in detail and illustrated on a concrete example of FPGA device. The design space exploration of an application deployment using this model is also presented.

1. Introduction

The demand for high performance applications has led to the development of heterogeneous multiprocessor system-on-chips (MPSoCs) which support both programmable processors and fixed hardware modules (dedicated hardware IP cores) as processing components. These systems can often benefit from the use of dynamically reconfigurable logic to combine application dynamicity and flexibility with high performance. The reconfigurable resource is typically an FPGA embedded in the MPSoC (eFPGA), and the resulting system is called multiprocessor reconfigurable system-on-chip (MPRSoC).

The eFPGA can be used to support parallel and/or sequential instantiations of different hardware tasks within the same area of silicon [1]. With dynamic reconfiguration, reusing parts of the custom logic allows to significantly reduce the area costs. It is thus necessary to analyse different task implementation variants and combinations in the reconfigurable logic. This new implementation space impacts the design space exploration (DSE) challenge of MPRSoCs, making it more important to have good tools available for exploring the different options and controlling the complete design flow.

The development of models at a sufficiently high level of abstraction helps in solving this problem. However, the level of abstraction must be properly chosen. It must be low enough to permit the estimation of performance and high enough to avoid considering artifacts irrelevant for what the modeling activity intends to achieve. In addition, in an optimal design flow, the level of abstraction must also support model transformation and code generation to simulate

and/or synthesize different parts of the system. The model driven engineering (MDE [2]) approach has been developed to provide efficient methods and tools to manage the design of complex systems. Software design was probably the first domain to use MDE, and hardware design is now also being considered.

In the context of embedded systems, power consumption is one of the most important issues. This leads to several other constraints like processor speed (faster processor consumes more power) and memory size (more memory also needs more power and requires more space) whose effects on a complex architecture are difficult to predict. There is currently no abstract model for estimating the performance and power of systems including an embedded reconfigurable element. Such methods and tools are necessary to help designers verify the system's characteristics (performance, power, etc.) especially at early stages of the design.

The work presented in this paper makes several contributions in the development of a global MPRSoC design flow. It is based on research conducted in the framework of a project called Open-PEOPLE (Open-PEOPLE: http://www.open-people.fr/) [3] gathering the expertise of several French academic and industrial partners in the field of low-power design. This project addresses the measurement, estimation, and optimization of energy consumption in complex MPRSoCs through high-level modeling of the components of the system. It proposes to build a framework allowing to control and measure the power consumption for different embedded platforms and applications. The methodology which is developed aims at easing the definition of efficient power models from these measures and developing tools exploiting these models to optimize power consumption at a high level of abstraction.

The AADL modeling language has been chosen for this purpose. This language was developed for the needs of performance critical real-time systems, initially in the field of avionics (Avionic Architecture Description Language), thereafter extended to general embedded systems (Architecture Analysis and Design Language). However, while AADL is suited to model generic homogeneous and heterogeneous multiprocessor architectures, it is not explicitly designed to model reconfigurable devices embedded in such systems. This problem is critical for MPRSoC design because complete DSE of these systems cannot be ensured without models of the reconfigurable logic.

The contribution of this paper is to present an extension of AADL supporting the modeling of reconfigurable logic, with the aim of enabling design space exploration of complex MPRSoCs including a reconfigurable element and considering power consumption requirements. The proposed AADL extension is based on a multilayer approach for specifying the system at three distinct levels of abstraction. At the highest level, the first layer is defined to capture general characteristics common to any FPGA. A second layer refines this view by capturing the static (non configurable) part of a specific device or FPGA family. Finally, a third layer is used to specify the actual use of FPGA resources, this is where the deployment of application tasks takes place. The remainder of the paper is organized as follows: Section 2 presents the state of the art in model driven engineering (MDE) and describes tools supporting MDE methods; Section 3 introduces AADL with enough details to understand the extension proposition to support the complete description of an FPGA within a SoC (Section 4); Then, an illustration of how the extension can be used is presented in Section 5, and an example of FPGA design exploration is detailed in Section 6; A discussion is proposed in Section 7 to compare our proposal with some other works; Finally, Section 8 concludes the paper and presents future works.

2. State of the Art

Significant efforts have been dedicated to high-level modeling during the past years. This is mainly due to the important growth of systems complexity. This complexity is more and more difficult to manage and new levels of abstraction are required. One interesting answer to this requirement was the development of the model driven engineering approach. MDE aims at representing systems at high levels of abstractions and proposes tools and methods to help the designer to verify, validate, and transform his specification earlier in the design flow. In this section, several MDE methods and languages are briefly presented. These methods can be classified according to their types and domains of application (software and/or hardware domains).

2.1. Languages for High-Level Modeling. The first domain is software development for which the unified modeling language (UML) is the most significant language. UML provides several diagrams to express different views of the system under design [4]. Depending on the design step, different types of diagrams are used for the definition of each part of the software. While UML is well suited for classical software design, a widespread criticism is that it is not well adapted for hardware modeling.

To solve this problem, the MARTE profile for embedded systems [5] has been proposed as an extension of UML. It is an interesting approach for complex SoCs containing both software and hardware components. MARTE enables to model the software and hardware parts of a system and to define the mapping of software components onto hardware components with an allocation model. Recently, this proposition has been extended to model multiprocessor system-on-chips. Several extensions have been also proposed to include reconfigurable aspects in SoC [6]. The extensions define a new stereotype for the FPGA component and extend some of the existing MARTE stereotypes such as (*HwProcessor*), memory (*HwMemory*), bus (*HwBus*), and arbiter (*HwArbiter*) by providing new attributes for dynamic reconfiguration. However, the dynamic aspects of reconfiguration are not completely covered due to the impossibility to express the physical layout and topology of the architecture.

In [7], the authors present an UML extension for exploring the allocation of tasks onto reconfigurable systems. The system is based on reconfigurable coprocessors strongly coupled to processors. The dynamicity of reconfiguration is supported by the reconfiguration of each separated

coprocessor, but the partial reconfiguration is not supported within one specific co-processor but between a set of co-processors.

SysML, which is an extension of UML, is a general-purpose graphical modeling language for specifying, analyzing, and verifying complex systems. SysML may include hardware, software, information, procedures, and facilities [8]. The language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parameters which are used to cooperate with other engineering analysis models. Performance analysis can be performed to verify that the application satisfies predefined constraints. However, SysML remains a general language and is not especially designed for the embedded system domain.

xMAML [9] is an extension of the machine markup language (MAML) [10]. It introduces new concepts to model dynamically reconfigurable architectures. While the language seems to have a good coverage of the domain, it has never been standardized and is not suited for the description of general embedded systems. Thus, if it were to be used in a general embedded system design process, integration with a standard language such as MARTE or AADL would be needed.

AADL allows modeling both software and hardware parts of digital systems. This modeling can be done by textual description or through graphical tools. The deployment of software components (processes, threads, and data) can be defined over hardware components such as processors, buses, devices, and memories. AADL has also advanced concepts for the modeling of alternative operational states of a system (modes) and the detailed description and analysis of information paths through components (flows). Languages for the description of component behavior, and for the specification of redundancy management and risk mitigation methods, have also been developed and added to the core language as standardized annexes.

2.2. MDE Approaches Based on These Languages. Each of the previously presented languages has its own domain coverage capability and typically model transformations can be developed to derive a level of description towards another [11]. Among these languages, only AADL and MARTE are dedicated to embedded systems. For both languages, methods and tools have been developed to support code generation for the targeted implementation platform. AADL is supported by a comprehensive tool suite providing efficient model analysis [12, 13], verification [14, 15], simulation [16], and code generation capabilities [17]. MARTE not only has the advantage of benefiting from all existing UML tools (at the expense of inheriting all UML features), but also from other tools such as the Gaspard2 environment distributed by the dart team [18]. Gaspard2 eases the modeling of SoCs by using specific structures like the repetitive model of computation, which simplifies the description of all the repetitive structures of the application or architecture elements. The tool enables different analyses including scheduling.

As previously mentioned, MARTE is implemented as an UML profile while AADL is a domain specific language (DSL), which, by definition, intends to be the most appropriate formalism for the targeted domain. As opposed to general modeling languages such as UML, which are polluted by several constructs defined to fit a larger domain, a DSL minimizes accidental complexity by retaining only the concepts of the specific domain. A good example of this can be found in the MARTE profile where nothing prevents a class having the processor stereotype to inherit from a class having the bus stereotype. Such construct is obviously wrong since a processor can never be a bus, and the only way to prevent this is to add an external constraint to the metamodel. Similar error cannot exist in AADL because of the language structure and fewer external constraints are required for the metamodel.

On the other hand, MARTE has the advantage of providing the designer with a larger set of fine-grained built-in concepts (stereotypes) than AADL. For example, a device in MARTE can be a generic device or a specific device such as an actuator, a sensor, and so forth. In addition, the software resource modeling (SRM) package already contains profiles and stereotypes for the modeling of real-time operating systems (RTOS). In AADL, the operating system is typically embedded in the processor concept, which is defined as "an abstraction of hardware and associated software that is responsible for scheduling and executing threads" [19]. Detailed analysis of the OS based on AADL models would eventually require the addition of user extensions to model the OS services at the proper level of granularity. Nevertheless, AADL appears to be a better choice to model the reconfigurable devices in order to verify and explore the design space due to its built-in mode abstraction that has no equivalent in MARTE.

3. Overview of AADL

AADL is a modeling language which allows using separate models to define applications and architectures. The language enables to define a global exploration and design flow as presented in Figure 1. This flow shows three parts which are the following:

(i) part (a) presents the AADL- modeling aspects that are independent from the hardware platform;

(ii) part (b) presents the modeling activities enabling the analysis and optimization phases along with design space exploration;

(iii) part (c) shows the code generation phase targeting specific execution resources;

The first models to define when using AADL are the application and architecture models. These models are defined through the three categories of components proposed in

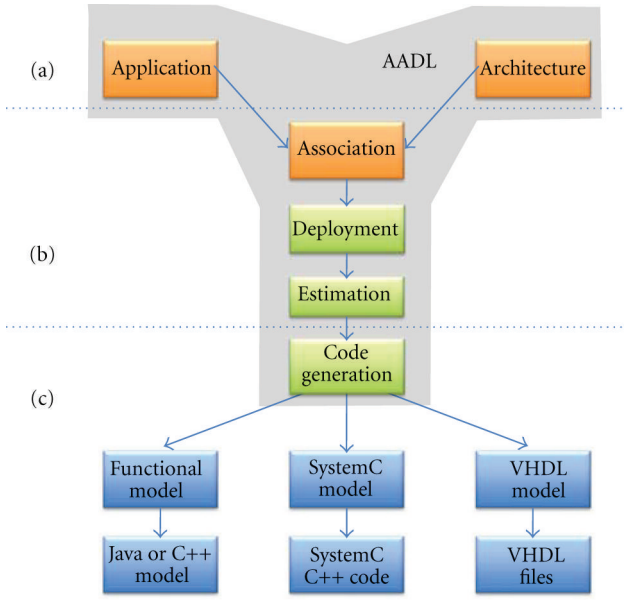


Figure 1: Using AADL (a) to model (b) analyse and (c) generate code for complex MPSoC.

AADL: software, execution platform, and composite. For each category, AADL proposes the following sub-categories:

(i) `application (software)`

  (1) `process, thread, thread groups, subprograms, data`

(ii) `execution platform (hardware)`

  (1) `processor, memory, bus, device`

(iii) `composite (hybrid)`

  (1) `system`

In AADL, components are defined via type and implementation declarations, which can be organized into packages. Component types can only declare externally visible features like interfaces in the Java programming language. On the other hand, implementations can precisely define the internal composition of components with the help of subcomponent declarations. Component types and implementations can have properties, whose types are defined in property definitions that are organized into property sets. Types and implementations can be extended so that features and properties of the parent component become visible from the extending component. New features and properties can be added to the extending component, and inherited features can be refined. Besides, advanced concepts exist, such as `modes` for the modeling of alternative operational states of a component, and `flows` for providing the detailed description and analysis of information paths through components.

As already mentioned, AADL is well suited for the description of processor-based systems. Monoprocessors, homogeneous, or heterogeneous multiprocessors can be described without difficulties. Memory organizations can be modeled by defining several memory levels (like data and/or instruction caches). Each component (software or hardware) can be characterized by dedicated properties that can be used to perform various analyses. For example, the execution time of a `thread` can be expressed with a `Compute_Execution_Time` property, and after deployment on a `processor` component, it is possible to verify allocation constraints on the processor. The open source AADL tool environment (OSATE) contains built-in plugins implementing this type of analyses.

When the architecture contains a complex reconfigurable device such as an FPGA, the AADL standard execution platform components are not directly usable. Indeed, an FPGA is very different from a processor. In particular, the processor's resource-based properties have no natural correspondence with those of an FPGA.

Fortunately, AADL is an extensible language, which is essential for meeting project- or domain-specific requirements. Two extension mechanisms are provided. A first one consists in the addition of user-defined property sets and packages declaring new properties and component types and implementations to constitute a so-called user extension. These are to be used as libraries for the modeling of other systems. The second mechanism is the definition of new sublanguages in the form of annexes to extend the semantics of the AADL core. However, an annex must be defined with great care and approved as part of the language by the SAE AADL subcommittee. This is not always an easy process. Fortunately, as will be shown in the next section, the first mechanism turned out to be sufficient to model an FPGA.

4. Extending AADL to Model Classical FPGA and eFPGA SoC Elements

In this section, the AADL extension dedicated to the modeling of generic FPGAs is presented. New classifiers are introduced for modeling the FPGA as a composite device (system) whose content is made of predefined (static) and synthesizable subcomponents. Dedicated properties are introduced to capture aspects that are relevant for any FPGA whatever its model or manufacturer. In this context, an FPGA may have two main zones:

(i) An area that can be configured to support the execution of specific IP blocks through a reconfiguration signal port (which corresponds to the ICAP port within the Xilinx V5 and V6 families).

(ii) A static area including predefined hardware components such as processor cores, hardware blocks for peripherals, and so on. These components are always present and are not configurable.

The proposed approach to model a FPGA is based on a multi-layer model where each layer describes the FPGA at a given level of abstraction. First, a generic and high-level reconfigurable resource model is defined to capture aspects that are common to all FPGAs (generic level in Figure 2(a)).

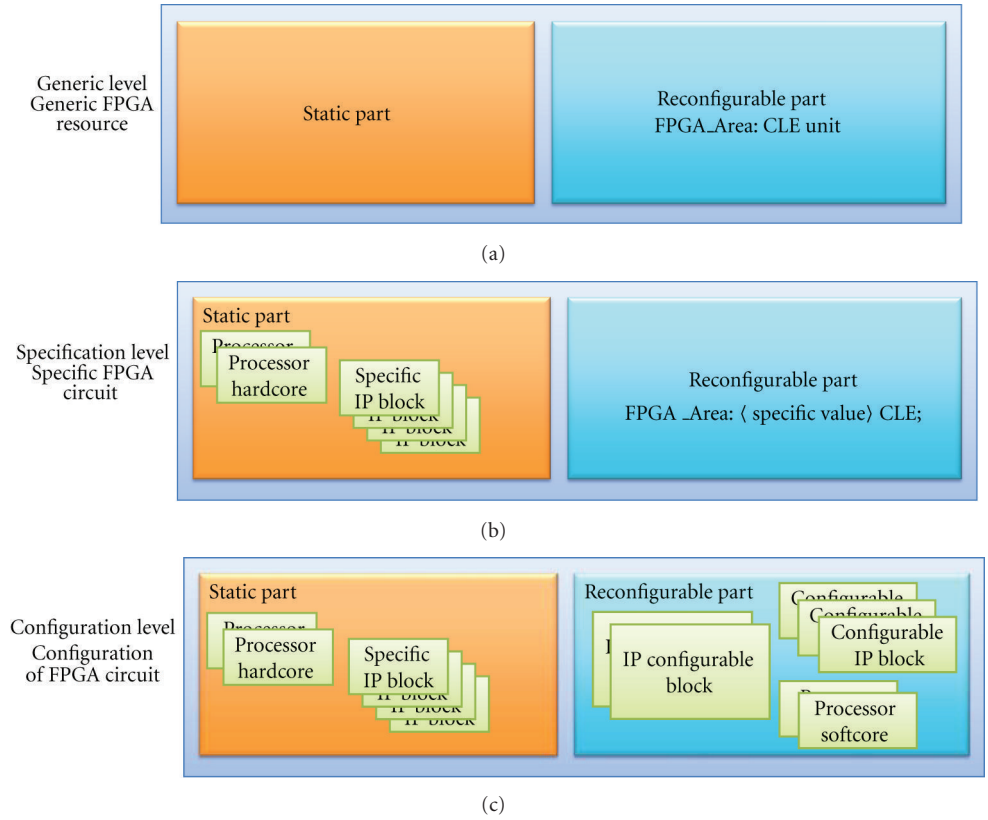


Figure 2: Multilayer and hierarchical approach to model FPGAs with the AADL.

This model is then specialized by a second layer model for each circuit in order to define all the IP blocks that are present in the static part (specification level in Figure 2(b)). Finally, a third layer for the specific configuration of the FPGA is defined with all the IP blocks that will be configured during the execution (configuration level in Figure 2(c)). The major advantage of this approach is that it provides a set of distinct layers to place information at the right place according to its level of abstraction. It also greatly favors reuse. For a given FPGA model, the static information is defined only once in a sort of library model that is then shared by every model of the third layer defining a given configuration of the configurable zone.

Due to its reconfiguration capacity, one important characteristic of an FPGA is the size of its configurable area. As a first simple verification, the constraint that the total area size of the simultaneously instantiated IPs does not exceed the total area of the FPGA can be verified. This constraint can be expressed by

$$\sum_{i=0}^{N_{IP}} A_i \leq A_{FPGA}, \tag{1}$$

where N_{IP} is the number of IP blocks present in the FPGA, A_i the area of the ith IP block, and A_{FPGA} the total area of the FPGA. Obviously, this simple verification is not sufficient to ensure that all the IP blocks can be instantiated within the reconfigurable area of the FPGA because the shapes of the tasks are not taken into account. However, if this constraint is not verified, it is sure that the IP blocks cannot be simultaneously instantiated within the FPGA. This point will be developed in Section 6.

These considerations lead to the definition of AADL component classifiers (types and implementations) and properties to capture the generic characteristics of FPGAs. Several ways to model an FPGA with AADL could have been proposed, depending on the verifications that must be performed. For the total FPGA area constraint (1), it may not be necessary to model the FPGA as a self-contained entity (AADL system). A simple approach could consist in viewing the FPGA as an implementation technology (e.g., ASIC 90 microns). A technology property would then be set on all synthesized components and used to distinguish the synthesized components from static components, in order to be able to sum the total synthesized area. However, this is clearly not sufficient. Identifying the FPGA via a technology property would not allow for distinguishing among several FPGA devices of the same technology that would coexist in the same system. Furthermore, it would not be possible to define properties applying to the entire FPGA such as the available size and power consumption. Clearly, the concept of the FPGA as a physical device entity is missing in this approach.

The FPGA is therefore modeled as an AADL `system` component into which sub-components of various natures can be statically defined or synthesized. Indeed, in the AADL,

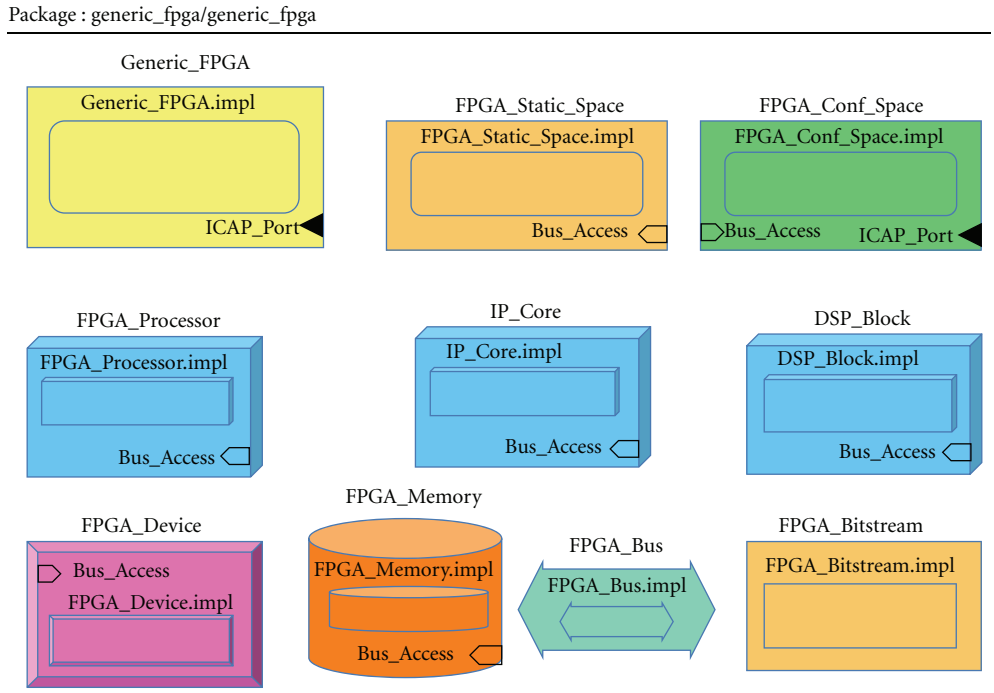


Figure 3: A diagram showing an AADL package declaring generic classifiers for FPGA components, and for all other component types that can be synthesized within the FPGA reconfigurable space.

only the `system` category can contain components of heterogeneous natures such as processors, buses, memories, controller devices, and so on. Choosing a `device` component would be wrong since it is defined as a black box whose internal composition cannot be specified. AADL component declarations are provided in the AADL extension for the FPGA itself and for its static and configurable zones (Figure 3). The static and configurable spaces are also declared as AADL systems. At a first glance, this may seem problematic for the configurable zone for the modeling of dynamical reconfiguration for which subcomponents may appear and disappear at runtime. However, as will be presented later in Section 5, the AADL core language `mode` construct turned out to be of great help to solve this problem.

Besides the FPGA and its two contained zones, AADL component declarations are also provided in the extension for every category of component that can be synthesized inside an FPGA (Figure 3). The component types are the following:

(i) three processor types to represent soft processor cores, dedicated IP blocks, and DSP blocks;

(ii) a memory type to represent memory components synthesized by connecting several BRAM blocks available in the FPGA;

(iii) a bus type for communication blocks within the FPGA;

(iv) a data type for modeling the bitstream data necessary to configure the FPGA. This bitstream may be partial in the case of dynamic and partial reconfiguration;

The reason for providing these declarations in the generic layer (highest level of abstraction) of the AADL extension is to have a place where properties specific to these types of synthesizable components can be attached. Indeed, since AADL V2 [19], properties that used were to be applicable to component categories can be restricted to user defined component types and implementations. For example, within these components, many of them can be characterized by the physical area that they occupy in the FPGA. To capture this information, and eventually other information pertaining to the FPGA AADL extension classifiers, a set of common properties is defined and added to the extension (listing of Figure 4). The area properties (`FPGA_Area` and `FPGA_Area_Required`) are expressed in terms of a number of configurable logic element (CLE) available in the FPGA. The `cle` unit, which is the elementary block of all commercial FPGAs, is used even if other units can be used among the various FPGA manufacturers. AADL allows the use of a set of units for a property and provides a mechanism to specify how the different units can be converted from each other. In the property set, two properties are defined in terms of CLE units; one for the total surface of the configurable zone of the FPGA (`FPGA_Area`), and another one for the occupied area of synthesizable components (`FPGA_Area_Required`). These distinct properties are meant to be applied to and only to the relevant classifiers: `FPGA_Conf_Space` for the `FPGA_Area property` and `FPGA_Processor, IP_Core, DSP_Block,` and `FPGA_Device` for `FPGA_Area_Required` property. Other properties such as the average toggle rate are also added to this property set with the same syntax for other verifications pertaining to this level of abstraction.

```
Property set FPGA_Properties is
  – Physical FPGA Area
  FPGA_Area: FPGA_Properties::CLE_Number_Type
    applies to (system);
  FPGA_Area_Required:
  FPGA_Properties::CLE_Number_Type
    applies to (system, processor, device);
  CLE_Number_Type: type aadlreal units
    FPGA_Properties::CLE_Number_Units;
  CLE_Number_Units: type units
    (cle, kcle => cle * 1000);
  ...
end FPGA_Properties;
```

Figure 4: A property set for generic FPGAs.

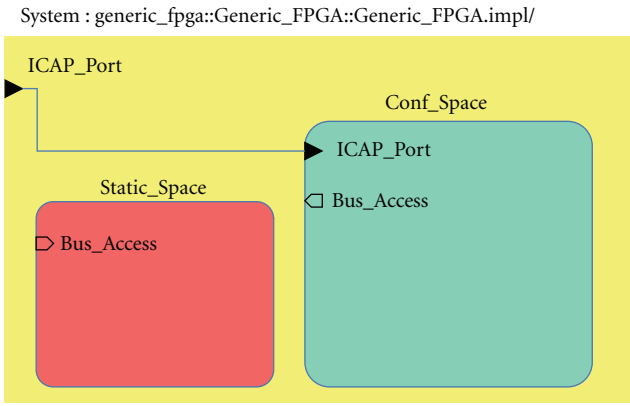


Figure 5: Graphical description of the composition of a generic FPGA with two main parts: static and configurable parts. The configurable part is connected to the configuration port which provides the dynamic and partial reconfiguration capability.

This first level of description is, however, not complete since the composition of the generic FPGA must be defined. The generic FPGA component implementation declares two system subcomponents representing the predefined static and configurable areas (Figure 5). Note that the types of the subcomponents are not defined at this level. As presented in the next section, both static and configurable parts are meant to be refined by the second and third layer model respectively, to provide declarations of static components always present in the given FPGA, and the hardware tasks that will be configured simultaneously and/or sequentially in the FPGA configurable zone.

5. Using the AADL Extension for a Specific FPGA Platform

From the AADL extension for generic FPGAs, a particular FPGA circuit can now be modeled. The FPGA circuit chosen in this part is the XC5VFX100T circuit, which is a Xilinx V5 FPGA embedding two PowerPC 440 processors.

As mentioned previously, the FPGA is modeled as a self-contained entity with a configurable zone into which subcomponents can be synthesized. The problem is that the number of ways in which the FPGA can be configured is obviously very high due to combinatorial number of possible configurations. The solution that we propose to address this problem is the following: first, define a model library to specify the non-configurable information (static space) of the specific FPGA; then, define separate models for each configuration of the configurable zone to be explored. These later models will extend the former static FPGA model library. In this way, the designer does not need to model the static aspects of the FPGA for every configuration he wishes to explore.

To define the model library, a package containing classifiers for the specific FPGA is defined. Figure 6 presents the XC5VFX100T model library that should never be modified by the designer once it has been created. This model captures all constant information of the specific FPGA such as the dual PowerPC composition of the static space and the total surface property of the configurable space (in this case 102400 configurable logic elements). A type (not shown) and an implementation are defined for the power PCs of the static zone (Figure 6(a)). A type and an implementation are also declared for the FPGA static space (`V5_Static_Space` in Figure 6(b)). They extend the classifiers of the generic FPGA extension. The implementation specifies the actual content of the zone, which is the two PowerPC subcomponents of the Virtex XC5VFX100T (Figure 6(b)). Classifiers are also declared for the complete FPGA component (`V5_FPGA`). They again extend the corresponding classifiers of the generic extension. The type of the `static_space` subcomponent is refined to the previously defined static `space` implementation (Figure 6(c)). As for the configurable space, the `V5_Conf_Space` classifiers also extend those of the generic extension, and a property association defines the actual size of the configurable zone (Figure 6(d)). It is obviously best to place this information in this library since it will never change for a given FPGA platform.

The next step is to model each configuration of the configurable zone that needs to be explored. For this, dedicated classifiers are defined (`V5_Conf_Space_1` in Figure 7). These classifiers extend the Virtex 5 configurable zone defined in the static library. In that way, the configured space component inherits the total area property value set in the library. The content of the zone is specified with the subcomponents declaration, where a BRAM memory, a PLB bus, an Ethernet controller, and an IP Core are synthesized (Figure 7(a)). Other classifiers are also declared for the completely configured FPGA (`V5_FPGA_Configured_1`) where the `conf_space` subcomponent is refined to the configured configurable space (Figure 7(d)).

After the modeling of the hardware part, the software application must be defined. In this example, a simple application is considered. This application consists in a single process containing four threads (`Basic_Process` in Figures 8(a) and 8(b)). Next, a global application system (`Basic_Application`) is declared, which contains the basic process as subcomponent.

The next step is to combine the basic application and the specific hardware platform declarations into a global system. This is modeled by the `Basic_Application_Deployed`

```
package xc5vfx100t
  ...
  Processor implementation PowerPC440.impl
    extends generic _fpga::FPGA _Processor.impl
  end PowerPC440.impl;
  ...
  – Implementation declaration for the static space
  – defining its specific composition.
  system implementation V5_Static _Space.impl
    extends generic _fpga::FPGA _Static _Space.impl
    subcomponents
      powerPC_0: processor PowerPC_440.impl;
      powerPC_1: processor PowerPC_440.impl;
    ...
  end V5_Static _Space.impl;

  – Implementation declaration for the FPGA extending the
  – generic FPGA of the AADL extension. The static space
  – is refined to the specific space of the V5 FPGA.
  system implementation V5_FPGA.impl
    extends generic _fpga::Generic _FPGA.impl
    subcomponents
      static _space: refined to
      system V5_Static _Space.impl;
  end V5_FPGA.impl;

  – Type declaration for the FPGA configurable space. It
  – provides a place to set the fixed Virtex 5
  – configurable space size.
  system V5_Conf _Space
    extends generic _fpga::FPGA _Conf _Space
    features
      icap _port: refined to
        in data port V5_Bitstream.impl;
      bus _access: refined to
        provides bus access V5_Bus.PLB;
    ...
    properties
      FPGA_Properties::FPGA _Area= > 102400 cle ;
  end V5_Conf _Space;
  ...
end xc5vfx100t;
```

a, b, c, d

Figure 6: Part of the AADL description of the hardware platform specializing the AADL generic FPGA extension (some details are not given in this description).

classifiers in Figure 9. The implementation contains subcomponents for the software application and the configured FPGA (Figure 9(a)). The deployment of threads onto processors and the process onto memory is specified with the help of predefined binding properties under the `properties` clause (Figure 9(d)). Finally, the number of CLE required by the synthesized IP_Block to execute `thread3` is set with the help of the `FPGA_Area_Required` property of the AADL extension.

So far, this is a typical AADL model, but from this, dynamic and partial reconfiguration modeling can now be presented. In this simple example, `thread3` can be bound to either the PowerPC 440 processors of the static zone or to a dedicated IP block of the configured Virtex V5 FPGA. To model this, the AADL `mode` construct is used. In the deployed application system, two operation modes are declared: a default initial mode characterized by a low power consumption (`low_power`) and another one for fast processing of data (`high_speed`), for which the power consumed by the FPGA would be significantly higher due to the synthesized IP allowing to speed-up the process (Figure 9(c)). Corresponding mode transitions are declared at this level and are triggered by the controller thread (`Controller_Thread` in Figure 8(c)) sending events at its ports which are connected to ports of the containing process and application system.

At the hardware platform level (`V5_Conf_Space_1.impl` in listing of Figure 7(c)), two modes (`ip_block_on` and `ip_block_off`) are declared for representing states where the IP block is synthesized or not. Mode transitions are triggered by events received on ports of the hardware platform and sequentially dispatched through the FPGA and its configurable zone. On the deployed application, connections are declared between the event ports of the application system and the ports of the hardware platform (`Basic_Application_Deployed.impl` in listing of

```
Package xc5vfx100t _conf _1
  ...
  – Implementation declaration for the configured FPGA
  – configurable space.
  system implementation V5_Conf_Space_1.impl
    extends xc5vfx100t::V5 _Conf_Space.impl
a   subcomponents
      bram_memory: memory BRAM.impl;
      plb_bus: bus V5_Bus.PLB
      ethernet _ctrl: device Ethernet _Ctrl.impl;
      ...
      ip_block: processor IP_Block.impl _1
        in mode (ip _block _on);
      hw_icap: processor xc5vfx100t::HW _ICAP;
b   connections
      data port icap _port -> hw_icap.icap _port;
      bus access plb_bus -> bram_memory.bus_access;
      ...
c   modes
      ip_block _off: initial mode;
      ip_block _on: mode;
      p_block _off -[ip _block _off _to_ip _block _on]
        -> ip_block _on;
      ip_block _on -[ip _block _on_to_ip _block _off]
        -> ip_block _off;
  end V5_Conf_Space_1.impl;
  ...
  – Implementation declaration for the configured FPGA.
  system implementation V5_FPGA_Configured _1.impl
    extends xc5vfx100t::V5 _FPGA.impl
    subcomponents
      conf _space: refined to
      system V5_Conf_Space_1.impl;
    connections
      data port icap _port -> conf_space.icap _port;
d     bus access conf_space.bus _access ->
        hard_space.bus _access;
      ...
      event port ip_block _on_to_ip _block _off ->
        conf_space.ip _block _on_to_ip _block _off;
      event port ip_block _off _to_ip _block _on ->
        conf_space.ip _block _off _to_ip _block _on;
  end V5_FPGA_Configured _1.impl;
  ...
end xc5vfx100t _conf _1;
```

Figure 7: An AADL package defining the dynamically configurable zone of the Virtex 5 FPGA.

Figure 9(b)). Note that each of these connections only exists in the proper system operation mode as specified by the `inmode` clause.

6. Exploring the FPGA Design Space with AADL

This section presents an example of early analysis of a SoC design including an FPGA. The analysis is based on a complete model of the system following the multi-layer approach and using the AADL extension presented in previous sections. The objective is to show how a more methodical analysis can be automated or semiautomated using high level AADL specifications and demonstrate how the proposed AADL extension can help exploring heterogeneous systems including reconfigurable units.

6.1. Use Case. The platform considered is composed of an embedded processor with a reconfigurable area (eFPGA). Indeed, many similar platforms are available today, like the Xilinx Virtex-5 FX Series FPGA which combines hard/soft processor IPs with programmable logic. Thus, our case study is composed of a PowerPC processor and a Virtex 5 FPGA. The application is described as a set of tasks with the characteristics presented in Table 1, assuming that each task can be mapped either on software or hardware. In these conditions, different combinations of task mapping using the implementation resources (PowerPC or FPGA) must be evaluated to provide good coverage of the design space.

In the following, we illustrate how the best solution can be extracted from different requirements such as FPGA resource usage, execution time constraints, energy and average power minimization, and availability of power management (processor idle states).

The application example is composed of three tasks that are supposed to be independent, periodic, and characterized

```
package basic _application
  ...
  – Type declaration for the basic application process.
  process Basic _Process
    features
a     low_power_to_high_speed: out event port;
      high_speed_to_low_power: out event port;
  end Basic _Process;

  – Implementation declaration for the basic application
  – process.
  process implementation Basic _Process.impl
    subcomponents
      thread1: thread Thread1.impl;
      thread2: thread Thread2.impl;
b     thread3: thread Thread3.impl;
      contr_thread: thread Controller _Thread.impl;
    connections
      event port
        contr_thread.low _power_to_high_speed
        - > low_power_to_high_speed;
      event port
        contr_thread.high _speed_to_low_power
        - > high_speed_to_low_power;
  end Basic _Process.impl;
  ...
  – Implementation declaration for the basic application
  – system.
  system implementation Basic _Application.impl
    subcomponents
      basic: process Basic _Process.impl;
    connections
      event port basic.low _power_to_high_speed
        - > low_power_to_high_speed;
      event port basic.high _speed_to_low_power
        - > high_speed_to_low_power;
  end Basic _Application.impl;

  – Type declaration for the reconfiguration
  – controller thread
  thread Controller _Thread
    features
      low_power_to_high_speed: out event port;
      high_speed_to_low_power: out event port;
c   properties
      Compute_Execution _Time => 28 Ms..32 Ms;
      Dispatch _Protocol => Periodic;
      Period => 100 Ms;
  end Controller _Thread;
  ...
end basic _application;
```

Figure 8: The AADL logical view for a simple software application containing four threads and using dynamic reconfiguration.

Table 1: Task characteristics for design exploration based on AADL description. The average power of the processor is supposed to be independent from the executed task and equals to $Pp = 100$ pu (power unit).

Tasks	Software version task			Hardware version IP block			
	Time Ts_i (tu)	Period Ps_i (tu)	Energy Es_i (eu)	Area Ah_i (cle)	Time Th_i (tu)	Power Ph_i (pu)	Energy Eh_i (eu)
T_1	20	100	2000	30000	10	100	1000
T_2	30	100	3000	40000	12	150	1800
T_3	60	100	6000	50000	20	250	5000

```
Package basic _application _deployed
  – Implementation declaration for the deployed application
  system implementation Basic _Application _Deployed.impl
    subcomponents
      application:      system
a       basic _application::Basic       _Application.impl;
      hw_platform:    system
        xc5vfx100t _fpga _conf::
          xc5vfx100t _Platform _Conf _1.impl;
    – Connections declarations
    connections
      event port application.low     _power_to _high _speed
        - > hw_platform.ip    _block _off _to _ip _block _on
b       in modes (low _power);
      event port application.high     _speed _to _low _power
        - > hw_platform.ip    _block _on_to _ip _block _off
        in modes (high _speed);
    – Modes declarations
    modes
c     low_power:initial      mode;
      high _speed:  mode;
    – Properties declarations
    properties
      – Thread binding
      Actual _Processor _Binding => reference
        hw_platform.v5p  _fpga.static    _space.powerpc  _0
        applies to application.basic.thread1;
      Actual _Processor _Binding => reference
        hw_platform.v5p  _fpga.static    _space.powerpc  _0
        applies to application.basic.thread2;
      Actual _Processor _Binding => reference
        hw_platform.v5p  _fpga.static    _space.powerpc  _0
        applies to application.basic.thread3
        in modes (low _power);
      Actual _Processor _Binding => reference
d       hw_platform.v5p  _fpga.conf  _space.ip  _block
        applies to application.basic.thread3
        in modes (high _speed);
      Actual _Processor _Binding => reference
        hw_platform.v5p  _fpga.static    _space.powerpc  _0
        applies to
        application.basic.contr        _thread;
      Actual _Memory_Binding => reference
        hw_platform.sdram  _memory
        applies to application.basic;
      – Area required by the synthesized IP
      FPGA_Properties::FPGA     _Area _Required => 50000 cle
        applies to
          hw_platform.v5p  _fpga.conf  _space.ip  _block
        in modes (high _speed);
  end Basic _Application _Deployed.impl;
end basic _application _deployed;
```

Figure 9: The software application deployed onto the configured hardware platform to constitute a PSM.

in terms of software and hardware execution time and energy consumption (see Table 1). The energy consumption is computed from the average power of the processor/hardware accelerators, and the execution time $Es_i = Pp \times Ts_i$ and $Eh_i = Ph_i \times Th_i$.

The hard IP processor is by definition represented using the static area of the FPGA and the hardware tasks with the configurable area, which supports partial reconfiguration. It is assumed that the maximum resource capacity of the configurable area is $A = 102400$ cle, and the nominal power consumption of the processor core is $Pp = 100$ pu (power unit).

In these conditions, different combinations of hardware and software executions are possible. At this level, AADL modeling allows to define tasks deployment on software and hardware components from which an analysis tool can verify the feasibility. For instance, the processor can not run all tasks without violating the period constraint of at least one task. On the other hand, the configurable area does not provide sufficient cle resources to implement all the tasks in hardware.

Each possible configuration that can be defined from the AADL model is presented in Table 2. This table presents the allocation of tasks and the corresponding global execution

Table 2: Example of possible combinations of task allocations within the execution platform when considering that the processor is always active even if no software task have to be executed. Note that the first and last rows are those which are not acceptable due to the total task execution delays ($T > Pp$) or the total task areas ($Ah > A$). Remarks: $Conf_1$ is incorrect due to the total execution time greater than the period of tasks; $Conf_8$ is incorrect due to the total area greater than the global area in the reconfigurable space.

Conf.	Task allocation		Software execution		Hardware execution			Global	Global without Low-power mode	
	Soft	Hard	Exec Time	Energy	Area	Exec Time	Energy	Exec Time	Energy	Av. power
			Ts (tu)	Es (eu)	Ah (cle)	Th (tu)	Eh (eu)	T (tu)	E (eu)	P (pu)
$Conf_1$	T_1, T_2, T_3	—	110	10000	0	0	0	110	10000	91
$Conf_2$	T_1, T_2	T_3	50	10000	50000	20	5000	50	15000	300
$Conf_3$	T_1, T_3	T_2	80	10000	40000	12	1800	80	11800	147
$Conf_4$	T_2, T_3	T_1	90	10000	30000	10	1000	90	11000	122
$Conf_5$	T_1	T_2, T_3	20	10000	90000	20	6800	20	16800	840
$Conf_6$	T_2	T_1, T_3	30	10000	80000	20	6000	30	16000	533
$Conf_7$	T_3	T_1, T_2	60	10000	70000	12	2800	60	12800	213
$Conf_8$	—	T_1, T_2, T_3	0	10000	120000	20	7800	20	17800	890

Table 3: Example of possible combinations of task allocations within the execution platform when considering that the processor can be in a low-power mode when no software task is executed. In this case, the power consumption of the processor is equal to 0.

Conf.	Task allocation		Software execution		Hardware execution			Global	Global without Low-power mode	
	Soft	Hard	Exec Time	Energy	Area	Exec Time	Energy	Exec Time	Energy	Av. power
			Ts (tu)	Es (eu)	Ah (cle)	Th (tu)	Eh (eu)	T (tu)	E (eu)	P (pu)
$Conf_1$	T_1, T_2, T_3	—	110	11000	0	0	0	110	11000	100
$Conf_2$	T_1, T_2	T_3	50	5000	50000	20	5000	50	10000	200
$Conf_3$	T_1, T_3	T_2	80	8000	40000	12	1800	80	9800	122
$Conf_4$	T_2, T_3	T_1	90	9000	30000	10	1000	90	10000	111
$Conf_5$	T_1	T_2, T_3	20	2000	90000	20	6800	20	8800	440
$Conf_6$	T_2	T_1, T_3	30	3000	80000	20	6000	30	9000	300
$Conf_7$	T_3	T_1, T_2	60	6000	70000	12	2800	60	8800	147
$Conf_8$	—	T_1, T_2, T_3	0	0	120000	20	7800	20	7800	390

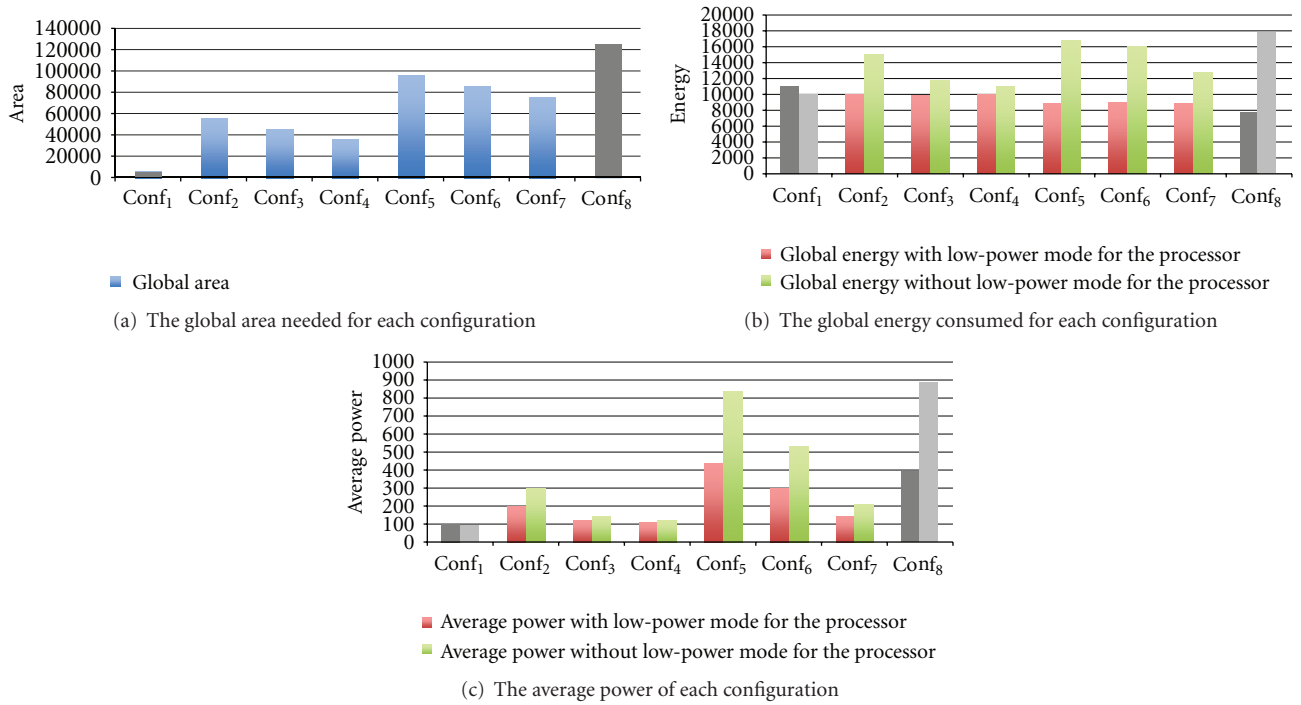
time along with estimations of energy consumption, for both the software and hardware point of views. Concerning software, if we consider that the processor is active, even if no task is running (no use of processor idle states), the energy of a task can be estimated from the average processor power consumption and the task period ($Es_i = Pp \times Ps_i$). The estimation of total software energy is the sum of the energy of each software task. Concerning hardware, each task exposes power numbers (Table 1) representing the average consumption of the associated hardware accelerator. We consider that all the tasks are executed concurrently in the reconfigurable area, so the global hardware execution time can be estimated by taking the maximum of all hardware tasks execution time. The estimation of total hardware energy is the sum of the energy of each hardware task (computed as $Eh_i = Ph_i \times Th_i$).

A global energy consumption estimation is also computed based on the sum of the hard/soft contributions. An average power consumption is simply derived from this energy and the global execution time: $P = E/T$.

Table 3 presents the same exploration example, but considering that processor idle states are available (e.g., suspend/resume). In this case, the processor can be put to sleep upon completion of software tasks, so compared to Table 2, only the software energy estimation (thus global energy and average power) is changing. If we neglect the delay and energy overheads resulting from switching between modes, energy of a software task T_i can be derived from its actual execution time and the average processor power consumption ($Es_i = Pp \times Ts_i$).

Figures 10(a), 10(b), and 10(c) provide graphical views that are derived from the two previous tables for better analysis convenience. We can easily notice the impact of task allocation on the design characteristics. The global area (Figure 10(a)) is computed by summing the contribution of each resource (processor, hardware tasks). In our modeling example, an amount of 6000 cle is associated to software for the implementation of processor-related logic (memory, bus, and peripheral controllers). Global energy (Figure 10(b)) and average power consumption (Figure 10(c)) are based on the results of Table 2 and Table 3, both when considering processor idle states or not, respectively.

Figure 10 shows that the global energy depends on the task allocation (deployment) on the execution target. In this



(a) The global area needed for each configuration

(b) The global energy consumed for each configuration

(c) The average power of each configuration

FIGURE 10: Results for the different configurations explored with AADL descriptions of a specific FPGA circuit and a set of tasks. As previously mentioned, $Conf_1$ and $Conf_8$ are incorrect due to their total execution time or total area.

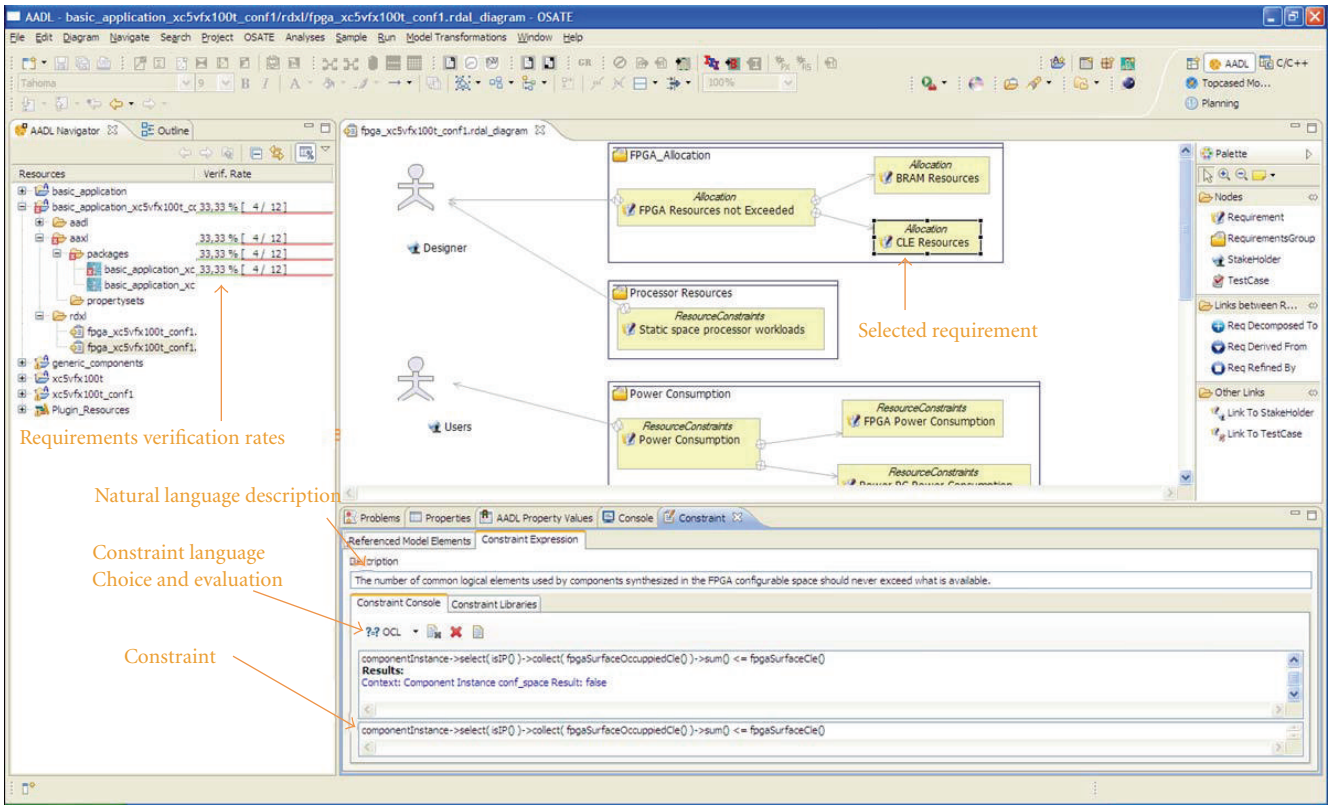


FIGURE 11: A requirements specification created for $Conf_1$ of the FPGA with RDALTE.

case, the configuration $Conf_1$ is incorrect because all the tasks are instantiated on the processor and the workload of the processor if more than 100%. Alike, the configuration $Conf_8$ is invalid because all the tasks are instantiated on the configurable area, and the total area is greater than 102400 cle.

From these results, it is easy to identify the best solutions for different types of constraints. For example, the most energy efficient solutions from (Figure 10(b)) are $Conf_5$ and $Conf_7$ when considering processor idle states, $Conf_4$ otherwise. For area and average power consumption (Figures 10(a) and 10(c)), the best solution is $Conf_4$ with estimations of 36000 cle (representing $36000/102400 = 35.15\%$ cle occupation) corresponding to 111 pu and 122 pu average power consumption, with or without using processor idle states, respectively.

6.2. Tool Chain. The main objective of the Open-PEOPLE project is to provide power analysis capability in a comprehensive model driven tool chain to ease design exploration of embedded systems modeled with the AADL. The tool chain is based on the eclipse IDE (integrated development environment) and combines various tools such as the open source AADL tool environment (OSATE), the ADELE AADL graphical editor, and the new requirements definition and analysis language tool environment (RDALTE). RDALTE and its language RDAL, which is currently being standardized by the SAE AS-2C subcommittee to become an annex of AADL, are both being developed in the frame of the Open-PEOPLE project.

RDALTE (Figure 11) allows to create requirement specifications attached to AADL models. RDAL requirements can be expressed with natural or formal languages of any type provided that an interpretation service for the language is available in the environment. At the moment, only OCL (object constraint language) has been implemented in RDALTE, but REAL (requirements enforcement analysis language [20]) should be available soon. RDAL requirements expressed with formal languages and attached to components of the AADL model can be automatically evaluated as the AADL design changes to provide traceability information telling to the designer which requirements are not met by the architecture model.

For the usecase of this section, RDALTE has been used in order to provide automated verification of the design space. A non functional requirements specification has been created (Figure 11) and attached to AADL models of the various FPGA configurations. Requirements have been created for representing constraints mentioned in the use case such as the number of used cle in the configurable space and the Power PC processor workload. As expected, the traceability views of the tool showed that $Conf_1$ and $Conf_8$ are not valid since they did not satisfy the processor workload and cle resources requirements.

7. Discussion

A few propositions are found in the literature to support the modeling of reconfigurable devices. In this section, a comparison with the approach of [7], which is based on MARTE is presented. The authors introduced UML extensions to the MARTE profile for exploring the allocation of tasks onto reconfigurable systems. The system is based on reconfigurable co-processors strongly coupled to processors. The dynamicity of reconfiguration is supported by the reconfiguration of each separated co-processor. Partial reconfiguration is not supported within one specific co-processor but between a set of co-processors.

A first difference between the two approaches is that they do not exactly target the same goal. In [7], the main objective of the modeling activity is to be able to generate the code for a given hardware platform. The intent is to provide the designer with an abstract view of the system, so that he can concentrate on functional aspects of his design. No technical information is attached to the hardware model. It is only when time comes to generate the implementation code that information on the actual hardware platform is given to the transformation tool chain. This is achieved by selecting the appropriate code generator into which the hardware platform specific information is encoded.

While this approach remains interesting, it is not well suited for verification and design space exploration purposes. In our case, our concern is to be able to perform non-functional verifications early in the design process, so that the designer can quickly make appropriate choices without having to perform code generation. For this type of analyses, the hardware platform must be known with a certain level of details to at least ensure that properties such as the area of the configurable zone are known. This is even more important for power consumption analysis, which typically depends on measurements on specific hardware components. In our approach, such verifications are made possible because the modeling considers the FPGA as a component itself, and not just stereotypes applied to synthesizable components as in [7]. In an ideal design space environment, exploration could even be broadened up to the FPGA platform level by providing a library containing AADL models for various FPGAs. The designer would then select a set of platform libraries, and, as was shown in Section 6, extend each of them to specify the content of the FPGA configurable zone. From there, design space exploration could start to help selecting the appropriate FPGA platform.

8. Conclusion

This paper presented an AADL extension allowing the modeling of FPGAs embedded in complex MPRSoCs. The proposed AADL extension is composed of property sets and packages and its use has been illustrated on a concrete FPGA example showing its applicability. This solution is intended to be used in a more global design flow targeting heterogeneous multiprocessor architectures including a reconfigurable element. In this context, the AADL extension allows considering the modeling of reconfigurable hardware blocks and power consumption at a high level of abstraction. This contribution is an important step towards the development of tools for the efficient design space exploration of these systems.

Although a manual design space exploration example was presented, it is based on detailed formal methods that can be easily integrated in an automated approach. This point is currently being studied in the Open-PEOPLE project

Future works will focus on the development of exploration heuristics specific to the dynamic and partial reconfiguration aspect. A first approach can be based on considering simply the size of the hardware tasks and sequencing the execution. In a second step, a more elaborate spatiotemporal scheduling algorithm considering the shape and position of tasks could be proposed. Suited AADL properties will be added in the FPGA model (position and size properties) to support this, and requirements analysis methods will be developed to exploit these properties. These perspectives will be applied to different types of FPGAs: Xilinx Virtex 5, Virtex 6, and future dynamic and partial reconfigurable from Altera.

Acknowledgment

The authors would like to thank the ANR (French National Research Agency) for supporting this work.

References

[1] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–6, 2006.

[2] "Model driven engineering, planet mde, portal of the model driven engineering community," 2008, http://planet-mde.org/.

[3] "The Open-PEOPLE Project Website," 2009, http://www.open-people.fr/.

[4] "Unified modeling language, omg," 2008, http://www.uml.org/.

[5] L. Rioux, T. Saunier, S. Gerard et al., "MARTE: a new profile RFP for the modeling and analysis of real-time embedded systems," in *Proceedings of the UML for SoC Design Workshop at (DAC '05)*, 2005.

[6] S. Cherif, I. R. Quadri, S. Meftali, and J.-L. Dekeyser, "Modeling reconfigurable Systems-on-Chips with UML MARTE profile: an exploratory analysis," in *Proceedings of the 13th Euromicro Conference on Digital System Design (DSD '10)*, Lille, France, September 2010.

[7] J. Vidal, F. De Lamotte, G. Gogniat, J.-P. Diguet, and P. Soulard, "UML design for dynamically reconfigurable multiprocessor embedded systems," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '10)*, pp. 1195–1200, IEEE, 2010.

[8] "Systems modeling language, omg," 2008, http://www.omg-sysml.org/.

[9] J. Lallet, S. Pillement, and O. Sentieys, "xMAML: a modeling language for dynamically reconfigurable architectures," in *Proceedings of the Digital System Design, Architectures, Methods and Tools (DSD '09)*, pp. 680–687, IEEE, Patras, Greece, august 2009.

[10] D. Fischer, J. Teich, R. Weper, U. Kastens, and M. Thies, "Design space characterization for architecture/compiler co-exploration," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01)*, pp. 108–115, ACM, New York, NY, USA, 2001.

[11] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[12] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *Proceedings of the Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies*, pp. 1–8, ACM, 2004.

[13] G. Harbour, G. Garcia, P. Gutierrez, D. Moyano et al., "MAST: modeling and analysis suite for real time applications," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pp. 125–134, IEEE, 2002.

[14] B. Berthomieu, J. P. Bodeveix, P. Farail et al., "Fiacre: an intermediate language for model verification in the topcased environment," in *Proceedings of the 4th European Congress on Embedded Real-Time Software (ERTS '08)*, Toulouse, France, 2008.

[15] B. Berthomieu, P. O. Ribet, and F. Vernadat, "The tool TINA—Construction of abstract state spaces for petri nets and time petri nets," *International Journal of Production Research*, vol. 42, no. 14, pp. 2741–2756, 2004.

[16] R. Varona-Gómez and E. Villar, "AADS: AADL simulation and performance analysis in systemC," in *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*, 2009.

[17] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the prototype to the final embedded system using the Ocarina AADL tool suite," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 4, pp. 1–25, 2008.

[18] E. Piel, R. B. Attitalah, P. Marquet et al., "Gaspard2: from MARTE to systemC simulation," in *Proceedings of the Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML Profile (DATE '09)*, vol. 8, 2009.

[19] "Architecture analysis & design language (aadl), version 2," January 2010, http://standards.sae.org/as5506a/.

[20] O. Gilles and J. Hugues, "Expressing and enforcing user-defined constraints of AADL models," in *Proceedings of the 5th UML and AADL Workshop*, UML and AADL, 2010.