A Model-Driven Approach for Reconfigurable Systems Development

Ismail Ktata1,2, Naoufel Kharroubi1

1Computer Science Department, Khurma University College, Taif University, Taif, 21944, Kingdom of Saudi Arabia
2CES laboratory LR11ES49, National Engineering School, University of Sfax, Sfax, 3038, Tunisia

ABSTRACT

Reconfigurable systems are considered as promising technology that enables the design of more flexible and dynamic applications. However, actually existent design flows are either low-level (so complex) or they lack support for automatic synthesis. In this paper, we present an ontology-based modeling approach for reconfigurable systems. Our approach is based on model-driven engineering process and addresses dynamic features on application-level enabling early exploration of the execution behavior of the system. The model is characterized by a logical and syntactical description conform with application domain knowledge and respect a number of metamodel constraints. These elements are semantically presented by an ontology language. We successfully implemented a system model with several tasks and resources and made scheduling test for the application graph.

1. Introduction

The system designers have been relying two choices for computing system implementing, varying largely in terms of performance and flexibility [1]. One is considered for the general-purpose systems built to accommodate flexibility problem. The compromise, in this case, is the system performance because of the absence of hardware resources dedicated for specific application. The second is carried out to develop an optimized hardware circuit dedicated and adapted to meet the demands of a specific application. Such system will be at issue following a minor need for change, because a costly redesign will be carried out in order to adapt new system parameters.

The present work introduces a new paradigm based on configurable computing that can solve performance problems as well as flexibility problems, and hardware design. We offer the performance of dedicated circuits by changing hardware configurations. The reprogram ability of the computing hardware explains that new architecture needs to emulate different computer architectures [2]. Mapping software operation is facilitated with getting the ability to reconfigure its connections. A number of manufacturers introduce FPGAs characterized by partial and dynamic reconfiguration abilities [3]. When a part of FPGA logic resources and interconnections is reconfigured while the remainder device continues operating, the partially reconfigurable FPGA is.

Dynamic reconfiguration deals with logic and interconnections reconfiguration in FPGA from an application to the next, that is what we call dynamic reconfiguration or run-time reconfiguration (RTR) [4]. In such situation, without adding external physical (hardware) resources, in a static or dynamic state at run time, when the system can change/adopt its basic computational structure to suit the changing requirements of a program, this is referred to as a Configurable Computing System (CCS) [5]. Our current CCS realizations has a general-purpose microprocessor augmented set with an FPGA. Using reconfigurable logic as a resource shared by multiple applications, a hardware operating system (HwOS) facilitates system setting operations to overcome the problems due to real time considerations. Real time multi-task systems’ requirement is tasks scheduling to complete their execution depending on their deadlines [6]. Tasks known periodic in advance are scheduled off-line. Instead, irregular and unknown tasks in advance are scheduled on-line. That makes specific constraints concerning computing resource recovery, preemption and interruption time, tasks’ migration between different executing resources, limited memory and power use, hardware costs, etc. [7]. To overcome such complexities, high-level development techniques are required. That researching methods dealing with modeling and simulation of complex and heterogeneous systems are called model driven engineering approaches.
Model-Driven Engineering (MDE) methods tackle with the specific constraints of a specific domain and execute model checking that may identify many errors and limit failure risks at an early period of the system's life cycle [8]. In this context, the present paper focuses on the use of an ontology model [9] in order to describe the dynamic behaviors of the components of a system within the schedulability analysis and verification context [10].

The proposed ontology is based on component behavior representation (Figure 1). Motivation behind our proposal is:

- System abstraction at a high level expressed in an ontology language enabling a prior evaluation and analysis of the impacts of possible online changes and reconfigurations.

- As a consequence of different multiple tools use, probable incoherencies may appear between design model and safety (normal) model.

- For architectural design, there is a heterogeneity in textual description, semantic representations, and syntax of the existing modeling languages.

- The need of reusable knowledge repository.

- Generating a safe executing model to be implemented.

For safety assessment, we elaborate, an ontology-based model description [11], [12]. We have several error behavior items that are stored in the ontology. Such reusable elements are a result of nominal models mixed with failure modes: each object is executable design system specification components under nominal behavior models. We transformed into formal safety models the obtained extended system models. A series of constraints (such as temporal and precedence constraints) are then added to the model in order to make it eligible to be analyzed and simulated by different tools, and so be reliable to making right decisions. All constraints are mapped onto an OWL domain ontology [13] written with the Protégé editor (Figure 2) [14]. Incorporated OWL description checked, the inference rules logic reasoned, and the ontology are able to detect mainly semantically inconsistent parts [15] as well as lack of model elements.

All errors interpreting cost are avoided: these errors encountered in the extended system model formal representation, are hardly to conceive. During the work, proposed architecture design is formulated in the Architecture Analysis and Design Language (AADL) [16], [17]. As a result, precise execution semantics are expressed for modeling software systems and their target hardware architecture. Safety models are generated in the AltaRica formal language [18]. We have opted for AADL and AltaRica languages, but we might as well have opted for other similar formalisms. The AADL language is adapted to describe functional and software architectures of embedded systems. The AltaRica language is an event-centric formal language suited for safety functioning modeling. Its translation is implemented in the form of model transformation. It would also have been possible to use other software such as ATL for example. There are series of analysis tools processed this language [19]-[21]. The proposed approach can be applied to validation contexts.

This work takes advantage of all previously cited domains: it is related to the scheduling problem in reconfigurable systems with a high degree of dynamicity. To overcome such complexity, we propose to work at a high abstraction level using an MDE approach combined with ontology description. The considered ontology language is OWL which gives a formal description of the database and the knowledge base and a semantic hierarchy of the whole system domain. In addition, after implementing the application constraints as rules, the ontology reasoning engine enhances the knowledge/data bases with new inferred axioms and data. All this is in order to have a perfect scheduling decision that takes into account all parameters/conditions and is flexible and dynamic according to the rule changes that may take place.

Related works are reviewed in Section 2. The domain ontology and the model-driven engineering process is presented in the Section 3. Section 4 is for the model transformation process and in the Section 5 we deal with the experimental case study. Then we conclude with a summary of the proposed process and we comment on its potential impact.

![Figure 1: Process of Ontology Engineering](image1)

**Figure 1: Process of Ontology Engineering**

The proposed ontology is based on component behavior representation (Figure 1). Motivation behind our proposal is:

- System abstraction at a high level expressed in an ontology language enabling a prior evaluation and analysis of the impacts of possible online changes and reconfigurations.

- As a consequence of different multiple tools use, probable incoherencies may appear between design model and safety (normal) model.

- For architectural design, there is a heterogeneity in textual description, semantic representations, and syntax of the existing modeling languages.

- The need of reusable knowledge repository.

- Generating a safe executing model to be implemented.

For safety assessment, we elaborate, an ontology-based model description [11], [12]. We have several error behavior items that are stored in the ontology. Such reusable elements are a result of nominal models mixed with failure modes: each object is executable design system specification components under nominal behavior models. We transformed into formal safety models the obtained extended system models. A series of constraints (such as temporal and precedence constraints) are then added to the model in order to make it eligible to be analyzed and simulated by different tools, and so be reliable to making right decisions. All constraints are mapped onto an OWL domain ontology [13] written with the Protégé editor (Figure 2) [14]. Incorporated OWL description checked, the inference rules logic reasoned, and the ontology are able to detect mainly semantically inconsistent parts [15] as well as lack of model elements.

All errors interpreting cost are avoided: these errors encountered in the extended system model formal representation, are hardly to conceive. During the work, proposed architecture design is formulated in the Architecture Analysis and Design Language (AADL) [16], [17]. As a result, precise execution semantics are expressed for modeling software systems and their target hardware architecture. Safety models are generated in the AltaRica formal language [18]. We have opted for AADL and AltaRica languages, but we might as well have opted for other similar formalisms. The AADL language is adapted to describe functional and software architectures of embedded systems. The AltaRica language is an event-centric formal language suited for safety functioning modeling. Its translation is implemented in the form of model transformation. It would also have been possible to use other software such as ATL for example. There are series of analysis tools processed this language [19]-[21]. The proposed approach can be applied to validation contexts.

This work takes advantage of all previously cited domains: it is related to the scheduling problem in reconfigurable systems with a high degree of dynamicity. To overcome such complexity, we propose to work at a high abstraction level using an MDE approach combined with ontology description. The considered ontology language is OWL which gives a formal description of the database and the knowledge base and a semantic hierarchy of the whole system domain. In addition, after implementing the application constraints as rules, the ontology reasoning engine enhances the knowledge/data bases with new inferred axioms and data. All this is in order to have a perfect scheduling decision that takes into account all parameters/conditions and is flexible and dynamic according to the rule changes that may take place.

Related works are reviewed in Section 2. The domain ontology and the model-driven engineering process is presented in the Section 3. Section 4 is for the model transformation process and in the Section 5 we deal with the experimental case study. Then we conclude with a summary of the proposed process and we comment on its potential impact.

![Figure 2: Protégé Framework Interface](image2)

**Figure 2: Protégé Framework Interface**
3. Ontology Domain and the Model-Driven Approach

We propose an engineering process to model a dynamic application mapped on configurable computing system. The process is based on (1) modelisation of the application to be created, (2) analysis and verification of the rules relating different models, and (3) test of the schedulability of the application from the resulting model. We assume that a CCS is computer-based system consisting of hardware and software components, integrated in a physical environment, and requiring different timing/ressource requirements on the final outcome.

3.1. Application Modeling Approach

We aim in this work to exploit domain knowledge in design models. We want to formalize the system domain design, its annotations and its knowledge in a single model. To do this, we followed a four-step approach (shown in Figure 3):

- The first step is to formalize and explicitly define the information (knowledge) of the domain of application in a formal ontology (entities, relationships, constraints, etc.). The properties of the ontology thus designed are domain-linked and completely independent of any context of use. In many cases, this ontology can be constructed from pre-existing standard ontologies.
- The second phase concerns the definition of design models. This means the formal definition of the properties corresponding to a given specification of the application domain.
- The third step, called annotation, is to define specific relationships between the different entities of the two previous models (design and ontology). Different relationships are available, they have their specific properties and describe specific rules related to the application domain.
- A verification step is required in any approach. This consists of validating the context of design models and domain properties expressed in the ontology, as well as the logic of the rules defined in the annotation step.
At the end, the final designed model has the advantage of well describing the domain knowledge with more details and properties derived from the ontology and rules generated during the annotation.

3.2. Scheduling Model

As mentioned above, we consider highly dynamic applications with soft real-time constraints. With such applications, missing a deadline does not cause catastrophic consequences, but only a performance degradation (quality of service parameter). In order to schedule these applications, we need a description that exhibits dependencies between the tasks and their own characteristics.

Each task is defined as a class in the ontology model. It represents a computational activity that needs to be performed according to a set of constraints (called slots in the ontology model). Each task $i$, for $i = \{1, 2, ..., k\}$ is defined by:

- $\xi$ represents a set of active tasks $T_i$ forming the application. Considered tasks could be sporadic and aperiodic.
- $a_i$: stand for the arrival time of task $T_i$.
- $C_i$: stand for the maximum computation (execution) time of task $T_i$.
- $C_i$: stand for the computation time of task $T_i$, i.e., the remaining worst case execution time WCET is needed for a computing elements (processor or logic circuit), at the current time, to complete the execution of task $T_i$ without interruption.
- $d_i$: stand for the absolute deadline of task $T_i$.
- $D_i$: stand for the relative deadline of task $T_i$.
- $S_i$: stand for the first start time of task $T_i$.
- $s_i$: stand for the last start time of task $T_i$.
- $f_i$: stand for the estimated finishing time of task $T_i$.
- $L_i$: stand for the laxity of task $T_i$.
- $R_i$: stand for the remaining time of task $T_i$.

Baruah et al. [32] present a necessary and sufficient test for synchronous tasks with pseudo-polynomial complexity. For that reason, we present the following equations defining relations among the parameters defined above:

\[ d_i = a_i + D_i \]  
\[ L_i = d_i - a_i - C_i \]  
\[ R_i = d_i - f_i \]  
\[ f_i = t + c_i; \quad f_i = f_{i+1} + c_i, \quad \forall i > 1 \]  
\[ R_i = d_i - t - c_i \]  
\[ R_i = R_{i+1} + (d_i - d_{i+1}) - c_i \] (7)

and, by equation (3), we have:

\[ R_i = d_i - f_i = d_i - (f_i - C_i) = d_i - (d_i - R_i) - c_i = R_i + (d_i - d_i) - c_i \] (8)

The information above present the major inputs of the task class ontology. In our model, we define the scheduling ontology basic classes as follows:

- Basic scheduling elements (including tasks, resources and activities) and relation between them.
- Basic required components and constraints (including hard/soft constraints, temporal restrictions and resource limitations).
- Ontology of instances: each class has one or more related subclasses.

The Protégé editor provides the definitions for basic object types and properties such as application graph (Figure 4), tasks set properties (Figure 5), relations, numbers, etc. Figure 6 shows the data properties of a created task refereed to its temporal features in the ontology framework. Figure 7 is related to processor creation, where processor is either a software processor or a CLB for hardware task. Figure 8 represents slots where defined the properties of a task, their status and needed executing resources.

We mean by resources all hardware components that are responsible for the execution of tasks. Resources could be a processor (for software tasks) or a discrete number of logic circuit called CLB (configurable logic blocks) for hardware tasks in an FPGA architecture. Resources are defined in separate classes and are characterized by their execution time and/or number of CLB.

Rules and constraints are modelled as distinctive class. The class constraint has the same definition for both hard and soft constraints, and is applied to tasks or resources. The hard constraints are the rules that cannot be violated under any conditions, while the soft ones have to be satisfied by the completion time of the scheduler. For example, if we consider the temporal constraints in the EDF (Earliest Deadline First) scheduler [33], the semantic rule would be:

\[ \text{Task } (\forall T_i) \land \text{has } D_i (\forall T_i, \forall d_i) \rightarrow \text{sqwrl:select } (\forall T_i) \land \text{sqwrl:orderBy } (\forall d_i) \]  

as EDF is a preemptive algorithm which assigns the highest priority to be executed to the task $T_i$ that has the lowest deadline $D_i$.

The both soft and hard constraints are applied to a task as well as a resource through the class schedule, which helps to satisfy schedulability conditions of the application execution.

3.3. Model Implementation

The Ontology-based model engineering architecture (Figure 9) is based on the Architecture Analysis & Design Language (AADL) and describes how the proposed ontology organizes the error models and the components into structural and functional interdependent hierarchies. AADL is considered as a textual
Figure 4: Application Graph Description

Figure 5: Task Class Properties

Figure 6: Task Creation With its Temporal Features
language and graphical one too. It is used to specify the software (SW) and hardware (HW) architecture for the needed critical level of real time systems [34].

In AADL language, each system is defined by its components. Each component identifies a number of elements of the actual system architecture. In our case, systems based on reconfigurable architecture are composed of a hardware part and software one. The AADL language permits to define software components (process, thread, data), as well as hardware components (device, processor, memory, etc.). Moreover, this design language defines precise legality rules that control the different component assemblies, and this in both static and dynamic (on-execution) way. To describe the communication between components, AADL defines the connectors and ports. A port performs a particular task in the context of the connector that connects components. All instances of connectors (including their ports and roles), compositions of components and components define the implementation operation. To enable linking flows to internal states, a model describes what initial states may evolve and it includes events, states, transitions to perform various analyses with different constraints on the model.

The objective of the design of the ontology is to prove the correlation of constructs related to the AADL specification, the retrieval, the storage operation’s permission, and to offer the desirable reasoning capabilities among them. We specified the ontology [33] in the OWL DL fragment of OWL and we applied Pellet DL reasoner combined with the Protégé ontology editor, and this to customize inference rules’ determination.

The proposed ontology is mainly based on three aspects [36]. Firstly, specify all necessary constructs for AADL core that permit to model components (e.g. data) with their corresponding characteristics and so facilitate ontology to add or extract new components. Secondly, for each AADL error, ontology offers the needed concepts to represent error models in terms of states and events, as hardware errors, computational problems and memory
exceptions, and thirdly, the ontology offers the connections and subcomponents (constructs) for component implementation. To achieve this task, we use ontology-based representation of component to detect rules custom inference [37]. The system detects three inconsistencies types making certain formal analysis models invalid: error of incomplete or missing transition; conflict transition that may occur when more than one event is triggered simultaneously; and finally, error in the state of a component due to an error in the failure scheduling scenario. The rules related to inconsistent semantics permit new inconsistent levels to be added. Meta-modeling design approach followed the ontology and determines instantiations and implementation steps. We represent the AADL component as an ontology class. The error models are presented as the ontology hierarchy subclass. The proposed model-driven engineering process enforces rules and constraints on the associations between the error models and the components’ constraints. It permits to select and resume the failure modes from the error model hierarchy, then associate them with the nominal mode. The ontology model is then checked for possible component inconsistencies and transform the extended architecture model to the safety one and finally analyze the safety model with tools that provide model checking and simulation.

4. Model Transformation Process

For discovering structurally equivalent constructs, existing modeling experience is considered a basic step for model transformation, and the first set of rules is dedicated to the transformation of AADL components into AltaRica nodes. The transformation rules are therefore driven by the mapping of knowledge with the constructs used through the underlying domain ontology. The benefit of the transformation from AADL models to AltaRica is to expand the set of safety evaluation tools for AADL. Hence, all system components (tasks, data, CLBs, processors, memory, etc.) are transformed to AltaRica nodes while conserving their same original features.

Then, AltaRica state statements are generated based on the AADL components’ properties. We manage a set of rules which concern the components’ error models. The AADL error, states and transitions are transformed to corresponding AltaRica which are filled with assignments found in the matched AADL. Another set of rules focuses on failure propagation. Since no support failure capabilities are made, a transition declaration creates additional component variants and other transformation rules set related to the used architecture design process and the associated AADL editor is made. The problem is that the editor tool follows a control-flow based approach, which make a semantic gap compared to the data-flow approach of the AltaRica specification.

An AADL component can be of two kinds from a safety point of view: either its properties are filled or are not filled [lost]. In the following, we will focus on the generic transformation. The names of the variables vary according to the connection ports of the component and the formulas of the component assertion depend on the data flow path. Indeed, any component of an AADL system:

\[ C = \{ D^{in}, D^{out}, K, Ty \} \]

is translated into an AltaRica node:

\[ N = \{ F, S, D^{in}, D^{out}, \Sigma, \sigma, I \} \]

where \( F \) is a field of finite values of the variables, \( S \) is for state, \( \Sigma \) is a set of events, \( \sigma \) is an affirmation function \( S \times D^{in} \rightarrow D^{out} \), and \( I \) refers to initial conditions, such as:

- the state variable \( s \) takes its \( \{\text{correct}\} \) value if the component is working normally otherwise its value is equal to the name of the failure mode \( \{\text{lost}\} \) declared in AADL;
- the dependency connections \( D^{in} \) and \( D^{out} \) form the AADL (and also the AltaRica) component interface;
- a failure transition is produced by events \( Evt \subset \Sigma \) leading to a failure mode;
- for each outgoing flow variable \( Out_j \in D^{out} \), we consider the set \( \{ In_i \mid (In_i, Out_j) \in D^{in} \times D^{out} \} \) of incoming flow variables on which the \( Out_j \) variable depends. Then, if we consider only the loss of a component, the statement associated with \( Out_j \) is the following \( Out_j = \{ \text{if} \ s = \text{correct} \ \text{and} \ In_i = \text{correct then correct, else lost} \} \);
- finally, we consider that, initially, components states are correct \( I(s) = \{\text{correct}\} \).

Considering an AADL system \( \psi = \{ D^{in}, D^{out}, \psi_1, ..., \psi_n, A, R \} \), \( \psi \) is transformed into an AltaRica node \( N = \{ N_0, N_1,...,N_n, V \} \) with:

- each subsystem \( \psi_i \) is transformed into an AltaRica \( N_i \) node;
- \( N_0 \) has for dependencies variables \( D^{in} \) and \( D^{out} \);
- \( R \) refers to connections between components and is equivalent to the \( \Sigma \) function;
- the synchronization vector \( V \) is given by the allocation relation \( A \) if a task \( \psi_i \) is allocated to a processor \( \psi_i \) then \( (evt_i, evt_j) \in V \) for \( e \in \Sigma \).

5. Experimental Case Study

We choose as a case study a 3D image synthesis application. It is a complex application with some flexibilities related to the computation time of some tasks and their executing occurrence. The application class hierarchy is implemented in OWL Protégé editor shown in Figure 10. The input of the application graph is a set of the coordinates of the different polygons’ summits that represent the 3D object. All coordinates are defined relative to a local space where the 3D object is located. Those coordinates are manipulated by different arithmetic functions (tasks) that create the animation, such as: Loading, Scaling, Adding, Rotating, Translating, etc. Tasks are implemented with their temporal features, and precedence relation according to the application graph. Rules are implemented by Semantic Web Rule Language (SWRL) interface (Figure 11) and combined with the scheduling ontology in order to improve the domain knowledge. Examples of some temporal rules are defined in equations (1) to (8), other rules related to allocation are like in (10) to indicate that processor can hold only one task to execute.
I. Ktata et al. / Advances in Science, Technology and Engineering Systems Journal Vol. 5, No. 6, 801-810 (2020)

Tasks(?T) ∧ hasLi(?T, ?L) ∧ Processor(?P) ∧ hasProcessor(?T, ?P) → sqwrl:select(?T) ∧ sqwrl:orderBy(?L) ∧ sqwrl:groupBy(?P) (10)

Figure 10: Class Hierarchy in Scheduling Ontology

After creating the SWRL rules, the reasoner component is able to infer the user query based on the ontology knowledge and predefined rules. To type queries, we tested two methods:

- The first method is simple and use existing information in the knowledge base without the need to any inferences. We use such method to verify existing properties in our ontology knowledge. For example, if we want to verify which processor is available for accepting a task execution, or the remaining computation time of some tasks.

- The second query method triggers the rule-based reasoner for the inference process with the knowledge which may lead to enhance and enrich the knowledge base. For example, if we consider tasks migration between processors or CLBs, and we want to decide migration of task T executing on processor1 when processor1 is overloaded. Hence, the rule-based reasoner has to infer the possibility of such migration based on the status of each processor and check that whether this change (so this query) is feasible or not.

By using AADL, nominal models were designed and combined also with failure modes of the domain ontology. The system’s adding process executes a simple add function, and it is defined in the Adding task. The operation is called performed when the required data context of the former task is provided to the data context of the latter task. To validate operation, scheduler check rules and constraint violation in the error model represented by the Overflow Event State, shown in Table 1.

Table 1: Example of incompleteness transition Error

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Error model</th>
</tr>
</thead>
<tbody>
<tr>
<td>error model implementation</td>
<td>error model OVERFLOW_2 Features</td>
</tr>
<tr>
<td>OVERFLOW_2_Notional transitions</td>
<td>Error_Free -&gt; OVERFLOW_2.Overflow;</td>
</tr>
<tr>
<td>Error_Free -&gt; OVERFLOW_2.Overflow;</td>
<td>Error_Free: initial error state;</td>
</tr>
<tr>
<td>end OVERFLOW_2_Notional;</td>
<td>Ovf_Temp: error state;</td>
</tr>
<tr>
<td>end OVERFLOW_2;</td>
<td>Over_Flow: error state;</td>
</tr>
</tbody>
</table>

6. Conclusion

We faced in this article the problem of modeling the complexity of reconfigurable systems while taking advantage of its flexibility and dynamic behavior. For this, we proposed a model-driven engineering process that follows a set of steps. The proposed modeling process makes a modular and extensible representation of the system architecture. Model profits from the ontology language capability to represent complex models and address heterogeneous domains information of the dynamically reconfigurable systems. We formalized the known temporal
scheduling problem in reconfigurable systems domain. The ontology knowledge model was implemented using Protégé editor. Moreover, the developed ontology is improved with SWRL inferred rules. Model editing is enhanced, since errors can be detected without the need to perform complex analyses. The model and the rules reasoner and the resulting scheduling decision were verified with a case study.

Future works aim to extend model transformation functionality and involve more dynamic system characteristics such as tasks migrations between hardware and software resources and the total and partial reconfigurability. A filtering process should be added to the reasoner in order to eliminate unnecessary rules and so avoid ambiguity or conflict when making scheduling decision. Another objective is to test more scheduling algorithms with the incorporation of additional types of error models.

Acknowledgment

This work has been approved by the research grant program of the deanship of scientific research at TAIF University.

References