Microprocessors and Microsystems xxx (2014) xxx–xxx

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A framework for reliability-aware embedded system design on multiprocessor platforms

Jia Huang [a,*], Simon Barner [a], Andreas Raabe [a], Christian Buckl [a], Alois Knoll [b]

[a] fortiss GmbH, Guerickestr. 25, 80805 München, Germany
[b] TU München, Boltzmannstr. 3, 85748 Garching bei München, Germany

ARTICLE INFO

Article history:
Received 5 July 2013
Revised 30 October 2013
Accepted 18 February 2014
Available online xxxx

Keywords:
Embedded systems
Reliability
Fault-tolerance
Design optimization
Model-driven development

ABSTRACT

This paper presents a model-driven framework that provides a tool-supported design flow for fault-tolerant embedded systems. Its system models comprise abstract descriptions of the application and the underlying execution platform. They provide the input to our analysis and optimization techniques that enable the automated exploration of design alternatives for applications with reliability requirements. The automated generation of source code and platform configuration files speeds up the development process. Our contribution is to advance reliability-aware design further into practice by providing an integrated tool framework and removing unrealistic assumptions in the analyzes. The case studies demonstrate the effectiveness of our approach.

1. Introduction

Reliability is becoming one of the most important concerns in today's embedded systems. However, as technology scales, modern devices are more susceptible to faults [5]. Such hardware faults could be permanent (hard errors) or transient (soft errors). Despite the efforts of the hardware community to enhance the hardware reliability, there is an increasing need to use system-level Fault-Tolerance Mechanisms (FTMs) to mitigate the impact of such faults.

Fault-tolerant embedded system design involves several challenging problems. First of all, the designer needs to reason about the system properties in the presence of FTMs to check if all requirements are met. Respective analysis techniques, such as reliability and timing analyses are needed. Second, since FTMs typically come with high overhead, it is critical to find the optimal design under given constraints. Therefore, a Design Space Exploration (DSE) problem arises. It is important to note that the configuration of FTMs must be considered jointly with other design parameters due to their correlation. In particular, the amount of redundancy highly influences the schedulability of the application. For multiprocessor systems, FTM configuration has to be considered together with the classical task mapping and scheduling problem. Finally, implementing the selected design on complex platforms such as Multiprocessor System-on-Chips (MPSoCs) is also challenging. Here, the designer faces both complex and labor-intensive problems such as multiprocessor programming, inter-core communication and the configuration of the execution platform. Therefore, tool support is highly desirable in order to reduce the design cost and the time-to-market.

Over the past decades, a lot of research effort has been devoted to the aforementioned challenges in the design of reliable systems. However, there is still a gap between theory and practice. On the one hand, due to the high complexity of the problem, the theoretical studies are typically based on certain assumptions and the according simplified system models. For instance, in many studies, it is assumed that any transient faults are detected at the end of the task using certain fault detectors so that all tasks have perfect fail-silent behavior [16,29,8,30]. Although a lot of important results have been achieved by studying this simplified version, such unrealistic assumptions limit the practical usability of the proposed techniques. On the other hand, most of the work focuses only on a single part of the overall problem and little effort is spent on integrated approaches providing tool support for the entire design process. In particular, current work mostly focuses on the "front-end" of the process, namely calculating the mapping/schedule under reliability constraints. The challenge of the "back-end", *i.e.* implementing the calculated schedule on a complex hardware platform, is underestimated.

This paper presents a model-driven development (MDD) framework to tackle the aforementioned problems. MDD is a well-known

* Corresponding author.
E-mail addresses: huang@fortiss.org (J. Huang), barner@fortiss.org (S. Barner), raabe@fortiss.org (A. Raabe), buckl@fortiss.org (C. Buckl), knoll@in.tum.de (A. Knoll).

http://dx.doi.org/10.1016/j.micpro.2014.02.007
0141-9331/

approach to cope with the rising complexity of embedded system design [26]. Our framework features modeling, analysis, optimization, and code generation tools in order to provide a complete integrated reliability-aware design flow. Fig. 1 compares the traditional development approach for reliable systems (left) to the design flow supported by our framework (right).

In the traditional approach, the designer extracts a scheduling model from the system specification in order to apply the reliability-aware scheduling algorithms. These algorithms may operate on different models (periodic task sets, task graphs, *etc.*) and the designer has to ensure the consistency. In parallel, the source code of the application is developed manually, including both functional code for the application and platform-specific structural code. Finally, the scheduling results and the source code are combined to an executable image and the platform configuration. The advantages of the proposed flow stem from the fact that models are first class citizens in MDD. They serve as central integration point for subsequent tasks such as analysis, DSE and code/configuration generation. Models speed up the development process and ensure the overall consistency by raising the level of abstraction and automation.

Our approach is based on the following:

- Platform-independent description of the application including timing and reliability requirements.
- Fine-grained platform model covering both hardware platform and system software (HW/SW stack).
- Multi-criteria design space exploration: mapping and scheduling w.r.t. timing and reliability constraints (consideration of permanent and transient faults).
- Automatic insertion of fault-tolerance techniques, including active redundancy, voting and fault detection to meet user-specified reliability goals.
- Generation of implementation artifacts such as application C source code and platform configuration files.

The remainder of the paper is organized as follows. The system models are presented in Section 3. The main contribution of this work, namely the reliability-aware design flow is presented in detail in Section 4. A case study is discussed in Section 5. Finally, Section 6 concludes this paper.

2. Related work

2.1. Fault-tolerant embedded system design

Using system-level FTMs has been addressed in a number of research studies. The approach presented in [36] handles transient faults by selectively inserting task re-executions. Girault and Kalla [8] consider fault-tolerant scheduling with active task replications and present a bi-criteria heuristic algorithm. Izosimov et al. [16] combine spatial and temporal redundancy and propose novel techniques to share the re-execution slack among multiple tasks. In the follow-up work [15], a more accurate probabilistic analysis is presented and hardware hardening is considered. Stralen and Pimentel present a DSE based approach for fault-tolerant deployment of applications on MPSoCs [33]. The FTMs are described as patterns that are applied to the application model. Only spatial redundancy patterns have been considered so far, *i.e., dual* and *triple modular redundancy* (DMR/TMR).

A recent work that is close to our approach is from Bolchini and Miele [4]. They also propose a generic DSE framework that supports a configurable set of FTMs, such as active redundancy, fault detection and voting. Moreover, they also synthesize time-triggered fault-tolerant schedules using genetic algorithms. One major difference between their work and ours is the fault model. They adopt a similar fault model as Izosimov et al. [16] and aim at handling a maximum number of concurrent faults. The reliability of the execution platform is modeled using a simple *qualitative* tag (*e.g.*, if the processor supports fault detection or fault-tolerance). Only coarse evaluation of the system reliability can be provided in this case. In contrast, our probabilistic reliability analysis provides precise *quantitative* results to guide the optimization process.

Tables 1 and 2 provide a qualitative comparison of representative related work. In Table 1, we first summarize the fault model utilized by the individual approaches. Some early work in the field [36,19] considers a single-fault model. This is a reasonable

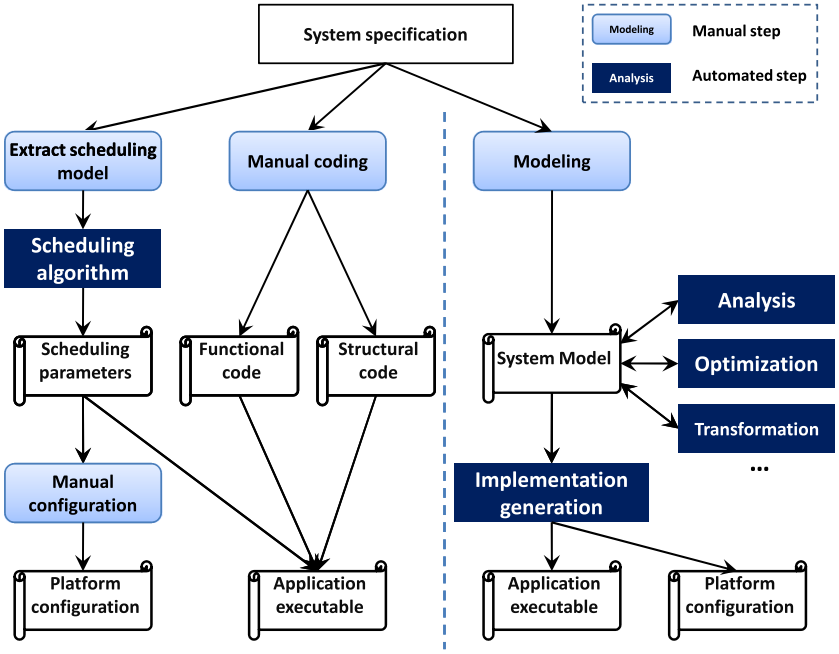

Fig. 1. Comparison of traditional and MDD flow.

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

Table 1
Qualitative comparison of related work: fault model.

Approach	Fault model		Reliability analysis
	Transient faults	Permanent faults	
Xie et al. [36]	Single	×	None
Glaßet al. [9]	×	Probabilistic	Quantitative
Girault and Kalla [8]	Probabilistic	×	Quantitative
Izosimov et al. [16]	Multiple	×	None
Izosimov et al. [15]	Multiple	×	Quantitative
Stralen and Pimentel [33]	Multiple	×	None
Bolchini and Miele [4]	Multiple	×	Qualitative
Proposed	Probabilistic	Multiple	Quantitative

simplification since faults typically occur with very low probability. These initial approaches have been extended with a fault model that covers multiple faults [16,15,4]. Still, probabilistic events are the most precise way to describe the physical properties of faults. The major challenge of using a probabilistic fault model is the complexity of corresponding reliability analyzes. For approaches based on a fault model that covers a certain number of faults, no detailed reliability analysis is needed, since the design objective is merely to add sufficient FTMs to tolerate all assumed faults. In contrast, a probabilistic approach needs quantitative evaluation of the system reliability after applying the FTMs. Recent approaches contribute appropriate reliability analysis techniques, but supports only very limited FTMs [8,15]. For this reason, we aim at providing a generic reliability analysis in our approach. Also, our framework is the only one that supports both transient and permanent faults. Table 2 lists the FTMs supported by individual approaches. As it can be seen, a major limitation of current approaches is that only a small set of FTMs is supported, making it infeasible to evaluate the tradeoff between various FTMs to find the system-wide optimal solution. Only the recent work presented in [4] and our approach try to support a *configurable* set of FTMs. The configurability enables the user to select candidate FTMs for the specific application domain and is therefore essential for the practical applicability of the approach.

2.2. Model-driven development

The UML profile MARTE [25], SysML [24] (based on a subset of UML 2) and AADL [7] enable modeling of complex HW/SW systems. While these approaches focus on the modeling aspect only, our framework also defines a design flow and provides an infrastructure for the implementation of platform-specific tool support (*e.g.*, modular code generator, modeling of HW/SW stack, *etc.*). DOL is closely related to our approach since it supports an entire design flow, including modeling, DSE and code generation [34]. However, our modeling framework is more generic and provides a variety of model-of-computations (MoCs) as well as platforms. DOL focuses on streaming communications since, so the application model is fixed to the Kahn Process Network (KPN) [18] MoC. Its platform model is tailored to shared-memory systems. Additionally, DOL does not consider reliability as one of the key non-functional properties and therefore does not provide the corresponding reliability analysis and optimization techniques.

3. System models

The proposed framework focuses on analysis and optimization of system-level reliability based on the reliability of the individual components. The component-level reliability is typically obtained using fault models derived from the physical failure mechanisms [35] and is configured by the user as an input to the framework. This section briefly describes the fault models used in our experiments.

For transient faults, we adopt the Poisson fault model since it is well established and used in many related approaches [32,1,8,37]. It assumes transient faults to be independent events following a Poisson distribution with a constant failure rate. Using the Poisson model, the following equations can be used to compute the success/failure probability of tasks:

$$P(\textit{task } t_i \textit{ executes correctly on processor } p) = e^{-\lambda_p w_i}$$
$$P(\textit{task } t_i \textit{ experiences transient faults}) = 1 - e^{-\lambda_p w_i}$$

where λ_p is the failure rate of processor p, and w_i is the Worst-Case Execution Time (WCET) of task t_i.

Concerning permanent faults, we focus on the defect of processing elements in MPSoC platforms. We assume that each individual core fails independently. Such fault containment behavior is essential to enable the use of MPSoCs for safety-related applications [23]. Based on given component reliabilities, the goal is to optimize the Mean Time To Failure (MTTF) of the system (c.f. [9]). Our DSE approach is designed primarily for transient faults. Nevertheless, permanent fault-tolerance is supported later on using an encoding technique in the optimization process (see Section 4.2).

We consider active redundancy as one major FTM to enhance the system reliability. Active redundancy replicates software tasks into multiple copies (replicas). The replicas can be executed on the same component (temporal redundancy) or distributed to several components (spatial redundancy). The availability of replicated software tasks allows for the implementation of subsequent voters where the set of available inputs is used to detect/mask possible faults and produce a reliable output. In this paper, we consider a **majority** voter, which generates an output if and only if more than half of the inputs have the same value.

Another FTM that we consider is fault detectors which are embedded into tasks. This could be done in hardware, software or using a combination [21]. Software-implemented fault detection typically involves transforming the original program into an instrumented version that adds the capability to detect transient

Table 2
Qualitative comparison of related work: fault tolerant mechanisms.

Approach	Supported fault-tolerant mechanisms				
	Spatial redundancy	Temporal redundancy	Fault detection	Voting	Other
Xie et al. [36]	×	√	×	×	
Glaßet al. [9]	√	×	×	×	
Girault and Kalla [8]	√	×	×	×	
Izosimov et al. [16]	√	√	×	×	Shared re-execution
Izosimov et al. [15]	√	√	×	×	Hardware hardening
Stralen and Pimentel [33]	√	×	×	×	
Bolchini and Miele [4]	√	√	√	√	
Proposed	√	√	√	√	Shared re-execution

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

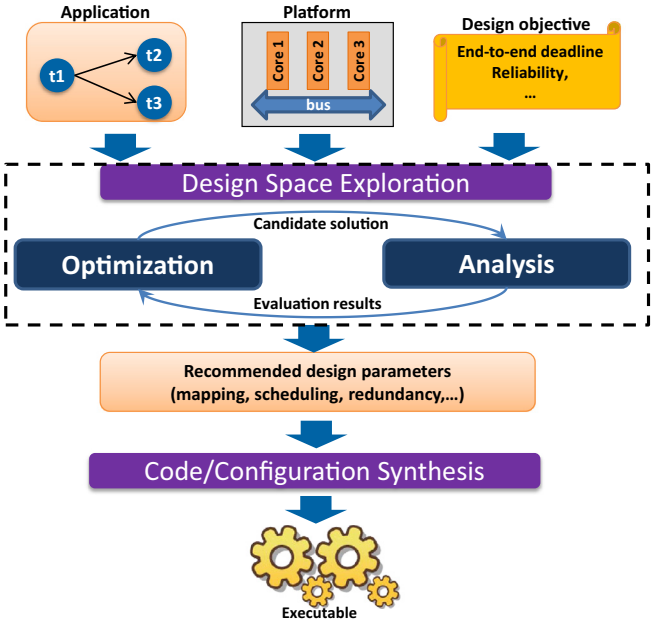

Fig. 2. Overview of the design flow.

faults occurring at runtime [31]. Check rules are executed at the tasks' completion to decide if faults have occurred. The arithmetic codes [31] and the critical variable technique [27] are examples of this class of detectors. Hardware techniques typically introduce some monitoring functionality. For instance, the fingerprinting mechanism [20] can be used to check if the program is executed as expected. The result of embedded fault detection is made available to the subsequent voter, so that untrustworthy results can be excluded from voting. This is similar to making the task fail-silent upon the detection of a fault. Using embedded fault detectors, the reliability analysis becomes more complicated since the fault detection coverage also becomes a factor that influences the system reliability. Also, the selection of the fault detector must be considered as an important design parameter.

Our approach focuses on heterogeneous multiprocessor architectures with time-triggered execution paradigm, since timing predictability is highly desirable for the targeted safety-related domains. In this case, the major objective of DSE is to find a time-triggered task/message schedule that fulfills the application requirements.

4. Reliability-aware design framework

Using the proposed framework, the design process is separated into three major phases: modeling, DSE, and code generation. An overview of the flow is depicted in Fig. 2. In the first phase, an abstract model of the application is created by instantiating elements of the meta-model library and specifying their relations. Additionally, a model of the execution platform is either also modeled or selected from a library. Lastly, extra-functional properties such as reliability of processors and the execution times of application tasks are specified. The DSE process takes the complete design model as input and aims at finding the optimal deployment of the application under user-specified design constraints (*e.g.*, end-to-end latency and reliability). We use the Multi-Objective Evolutionary Algorithm (MOEA) as a generic optimization engine. The result of the DSE is a set of recommended implementations. They are represented by updated models where the corresponding design parameters have been applied automatically. This includes the mapping of the application to the platform, the required computation and communication schedules as well as the addition of FTMs. In the final phase, the designer may select one of the implementations and have the generation engine produce source code and necessary platform configuration files. In the next sections, the details of each phase will be presented.

4.1. Model-driven development framework

Our MDD framework is based on the Eclipse Modeling Framework (EMF)[1] which supports the definition of meta-models and the automatic generation of tooling infrastructure code (*e.g.*, model serialization, Java implementation of meta-model, basic tree-editors, *etc.*).

The first part our framework is an *abstract component meta-model*. Its basic building block is the *Component* class whose external interfaces are described by *Port* objects. Systems are either constructed by linking the ports of components using *Channel* objects, or by encapsulating (sub-)models into composed components. Different application and platform modeling languages have been defined by sub-classing these three base classes, introducing dedicated types and attributes. An annotation mechanism provides an alternative possibility to enrich components, ports and channels with additional information. Since this approach requires the definition of dedicated annotation types, it lends itself towards properties that can be reused with different base models (*e.g.*, timing annotations that can be used with different application models).

In summary, the component meta-model is used to describe the "horizontal" relationship between objects residing at the same system layer. Examples include data or state flow models of the application, or the network topology of a distributed platform.

Fig. 3 shows is a simplified version of the model used in the case study in Section 5. We start with the discussion how the underlying application meta-models have been constructed using our framework, and will revisit the other meta-models later. The application model instance is shown at the top of the figure. In this example, the Kahn Process Network (KPN) [18] model-of-computation (MoC) has been selected to provide a platform-independent description of the application. It is a coarse-grained MoC that focuses on modeling the structure of the application and the interaction between components. KPN components represent computational tasks that communicate via FIFO buffers. The required meta-model has been constructed by deriving sub-classes for KPN components, ports, and channels from the abstract component model introduced above. The behavior of KPN components is specified using a dedicated annotation type for source code annotations (here: C code with markup for port access).

We also used our framework to construct more fine-grained (domain-specific) application meta-models from the generic component model. In particular, we defined meta-models for IEC 61131-3 [14] State Flow Charts (SFC) and Function Block Diagrams (FBD)/synchronous data flow (SDF). Here, SFC states and transition conditions are represented by dedicated component types (following the approach of [2]), whereas the composition of states (*i.e.*, sequence, alternative and parallel branches) is represented by SFC "channels". For FBD, dedicated meta-model classes have been defined that represent the function block semantics (*e.g.*, basic arithmetic or logical operations, controller functions, *etc.*). In contrast to black box components (such as the KPN components introduced above), this fine-grained approach allows for an explicit model of the application, *e.g.*, concerning its behavior, or its dependency on the execution platform (see below).

The second part of our framework provides a mechanism to model the "vertical" dependencies between the different layers

[1] http://www.eclipse.org/modeling/emf/.

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

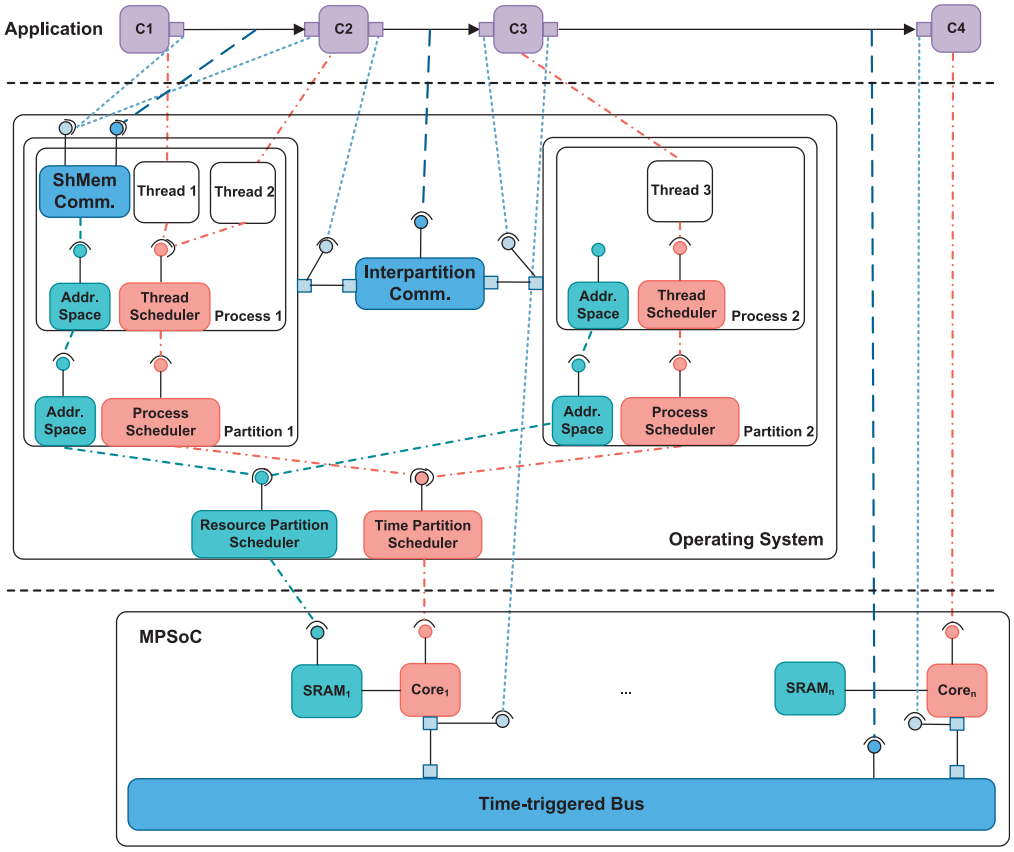

Fig. 3. System model (example).

of a system model. So far, only the top layer (*i.e.*, the platform-independent application models) has been discussed. In order to capture the resource usage of applications, and their dependency onto the execution platform, we have introduced a *resource meta-model* that provides a unified way to specify deployments (*e.g.*, the mapping of an application component onto a processor core, or the usage of an I/O pin by a component of a function block diagram). Since the resource meta-model decouples application and platform meta-models, it enables to reuse application meta-models with different platforms.

The bottom part of Fig. 4 shows the resource meta-model. *Resource* objects present system entities that provide services to other components in the system. They are typically located in the platform model. Each resource is annotated with an *arbitration* object that defines the arbitration policy used for accessing it. As policies, we currently consider *design-time assignments* expressing the exclusive use of a resource, *design-time schedulers* for time-shared resources with a statically defined schedule, and *runtime schedulers* for dynamic time-sharing. System entities that depend on services provided by the platform are described by *request* objects. Typically, elements of the application model are pure *request* objects, whereas models of a platform's software stack (*e.g.*, middleware, operating system) are typically both *request* and *resource* objects (since they rely on services provided by the lower layers of the platform, and provide a certain service themselves). During the deployment phase, requests are bound to resources. In the resource meta-model, the *Allocation* class is used to describe the fraction of a resource that is used to satisfy a given request. Typical examples include time slots (for time-shared resources such as a time-triggered bus), or spatial regions such as a certain address range of a memory resource.

In the following, we will sketch how the resource meta-model can used to extend the application meta-models introduced above (using multiple inheritance), as well as to construct the required platform meta-models. To this end, we will revisit the example model from Fig. 3. The bottom of the figure shows a model of the MPSoC platform from our case study (see Section 5). Its cores are interconnected by time-triggered network-on-chip, and have access to private memories. The middle part is a model of an operating system providing partitions for temporal and spatial separation. The resource meta-model does not provide any semantics about the different resource types used in a given system. Since this information is required to restrict the allocation of a given *request* of an application model to the set of matching *resource* objects of the platform, we propose to introduce a resource type meta-model. This meta-model represents a resource taxonomy that categorizes the resources of embedded systems (see top part of Fig. 4). We will use this categorization in order to structure the following discussion of the meta-models defined for our example system – representative examples of the meta-model classes are shown in the middle part of Fig. 4.

`Processing` abstracts computation services. The KPN components $C_1, \ldots, C_4$ in Fig. 3 are implemented in software that needs to be executed on a processor. Hence, `PnComponent` inherits from both `Request` and `Processing` (and `Component`). The hardware platform in the example model is an MPSoC consisting of several processor cores (inheriting `Resource` and `Processing`). During the deployment of a platform-independent KPN model to a concrete platform, the processing requests can be mapped to a dedicated core. Here C_4 is mapped to $Core_n$ which results in the allocation of the hard-thread (`HThr`, derived from `Allocation` and `Processing`) provided by the processor core. The middle part

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

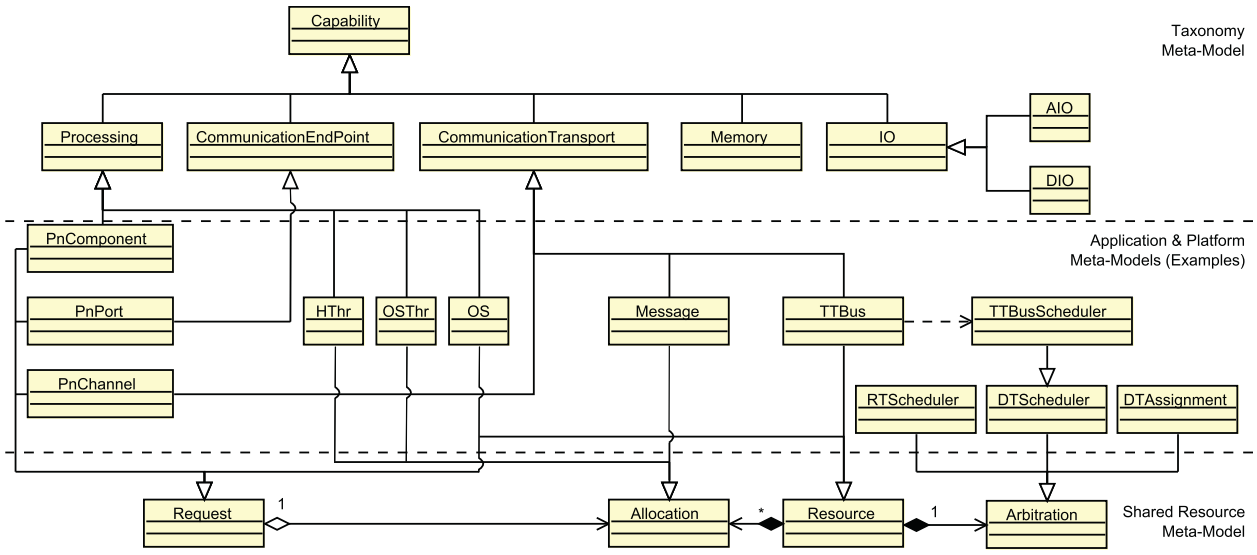

Fig. 4. Class diagram of application and platform meta-models.

of Fig. 3 illustrates how a separating operating system is modeled. Here, the processing request of the `OS` (super-classes `Processing` and `Request`) is mapped to $Core_1$. Additionally, `OS` also inherits `Resource` since the service of the processor is further split by its partition scheduler (not shown in Fig. 4). The same mechanism is applied recursively to define processes and threads at the OS level.

`Transport` describes platform components that can be used to move data in the system (*e.g.*, buses and networks at the hardware level, but also software communication APIs). In our example, a time-triggered bus (`TTBus`) provides core-to-core communication. Hence, it inherits from both `Resource` and `Transport`. It also references a `TTBusScheduler` object that contains the required design-time scheduler. In a KPN model, channels represent the data-flow between components. Therefore, `PnChannel` inherits from both `Request` and `Transport` (resource view), and `Channel` (topology view). The time slots that are allocated to these requests by the scheduler are represented by `Message` objects. In both cases, annotations provide additional information such as bandwidth requirements and scheduling constraints for channels or the message phase computed by the scheduler. In Fig. 3, also other communication resources are illustrated such as the partition-to-partition communication facility provided by the OS or thread-to-thread communication via a shared memory buffer (see below).

`EndPoint`: In addition to bandwidth, end-points at the sender and the receiver are required in order to fully specify a communication channel. In our example, ports of the KPN components (`PnPort`, derived from `Port`) request them from the platform. The platform provides end-point resources that match the transport resources described above.

`Memory` can be used to abstract volatile and non-volatile memories, as well as services realized in software such as file systems or databases. In Fig. 3, each of the processor cores is equipped with a local `SRAM` memory (super-classes `Resource`, `Memory`). Similar to the `Processing` service provided by the processor, the model of the operating system tracks how the memory is managed by the individual partitions and processes. The memory buffer that is used to implement the communication between the application components C_1 and C_2 contains a request to the address space provided by $Process_1$. Additionally, it is a combined end-point and transport resource to which the end-point requests of the ports of KPN components C_1 and C_2 as well as the corresponding channel are mapped.

In addition to the core resource types pointed out above, also more specific resources can be included into the taxonomy, leading to a more expressive system model. Typical examples are the `Dio` and `Aio` resource types which represent digital and analog I/O resources.

4.2. Reliability-aware design exploration

The DSE process is implemented as an optimization–evaluation loop. A probabilistic reliability analysis is used to evaluate the system-level reliability of the design. The results of the analysis guide the MOEA-based optimization. This section presents only the key ideas of the analysis and optimization algorithms whose implementation works on the system model introduced in the last section. A full description of the DSE framework is presented in [13].

4.2.1. Reliability analysis

Computing the system reliability in the presence of FTMs is a very difficult problem. Certain combinations of faults might be tolerable by the system due to the back-up components whereas other combinations are not. In principle, we have to identify all the tolerable cases in order to assess the system reliability. This has been proven to be at least as hard as NP-complete problems [3]. In our approach, a tree-based reliability analysis approach is proposed (*cf.* [11]), an example of which is shown in Fig. 5.

Our reliability analysis systematically enumerates all possible combinations of faults (called fault scenarios). Each fault scenario is represented by one node in the tree. Each level of the tree is associated with one component in the system (*i.e.*, one task in the schedule as shown in Fig. 5). We grow the tree from an initial node, where a left branch (solid line) represents a successful execution of the corresponding component and a right branch (dashed line) represents a fault which has occurred in that component. While expanding the tree, we analyze each new node and mark it if it represents a tolerable fault scenario (nodes C_1—C_4 in the example). In the end of the tree expansion, the set of all tolerable fault scenarios is obtained. Then, we compute the occurrence probability of each tolerable scenario and combine these probabilities to obtain the overall success probability of the system (see [11] for more details). The tree-based analysis especially fits our framework, since it is generic enough to be configured to handle different design scenarios. In our implementation, we support the analysis of designs

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

that employ spatial/temporal redundancy, voting and fault detection. Moreover, advanced techniques such as the shared recovery slot technique proposed in [16] are supported as well.

4.2.2. Optimization procedure

We encode the design into a special data structure called *chromosome* to use MOEA optimization. Since encoding the complete set of design parameters into the chromosome results in very large data structures and low optimization efficiency, we apply an efficient two-step encoding scheme inspired from [22]. The main idea is, instead of encoding the entire design, to include only partial information into the chromosome. A heuristic algorithm is used to derive the remaining design parameters to yield a complete design. Using our encoding scheme, the chromosome contains a list of mappings for each task as shown in Fig. 6. FTMs are inserted by adding special patterns into the chromosome. For example, temporal redundancy is represented by mapping a task onto the same component multiple times (*e.g.*, tasks t_2 and t_3) and spatial redundancy is represented by mapping a task onto different components (*e.g.*, task t_1). As we are considering time-triggered systems, the complete design is a time-triggered task/message schedule. We use a heuristic scheduler (*cf.* [11]) to transform the chromosome into such a schedule (shown in the left part of the figure). As it can be seen, the replicas specified in the chromosome are instantiated and voting components are inserted where necessary. The schedule is then submitted to the reliability and timing analyzer for fitness evaluation. Based on the fitness value, the MOEA optimizer uses mutation and crossover operators to explore high-quality designs.

An advantage of the two-step encoding is that additional application constraints can be taken into account by putting constraints on the chromosome. A typical example are separation constraints such as two critical tasks that are required to be strictly spatially isolated. This can be encoded as a constraint that the mapping entries of the two tasks do not collide. Another example is how we handle permanent faults. In current reliability-aware design approaches, permanent and transient faults are usually considered separately, since their physical failure mechanisms and impact on the system are significantly different. In [11], we show the benefit of considering both types of faults in a unified manner. One obvious way to achieve this is to integrate the existing analysis in [9] and consider lifetime reliability as one extra optimization goal. However, additional optimization objectives reduce the optimization efficiency considerably. To overcome this problem, we propose the virtual mapping technique and encode reliability

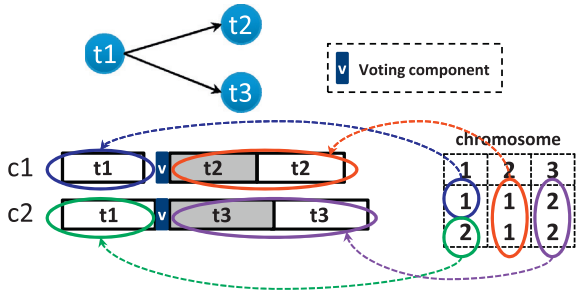

Fig. 6. Example of encoding technique.

requirements concerning permanent faults as constraints to the chromosome (*cf.* [11,13]).

4.2.3. Optimization goals

The MOEA optimization approach enables us to consider multiple (possibly conflicting) optimization goals in the design problem. Currently, we support several optimization constraints/goals that can be freely combined by the user: (1) end-to-end deadline; (2) reliability of the application; (3) resource consumption (processor time occupation); (4) application specific constraints, *e.g.*, a task must be mapped to a certain core due to I/O constraints. The MOEA algorithm evaluates the tradeoff between these optimization goals and computes the Pareto optimal results. The designer then makes the selection and proceeds with the implementation (see Chapter 4.4).

4.2.4. Complexity

The overall execution time of the MOEA algorithm increases proportionally with the number of iterations configured by the user. To guarantee the usability of the proposed approach, it is important to minimize (1) the per-iteration complexity, (2) the search space for MOEA algorithm. The per-iteration complexity of MOEA is determined by the complexity of the reliability analysis. However, the tree analysis algorithm presented previously has worst-case exponential complexity. For this reason, we developed safe approximation techniques to reduce the complexity to polynomial [11]. For the second issue, the hierarchical encoding technique contributes to reduction of the search space. With these techniques, experimental results verify the applicability of our framework for offline reliability-aware design (see results presented in [12]).

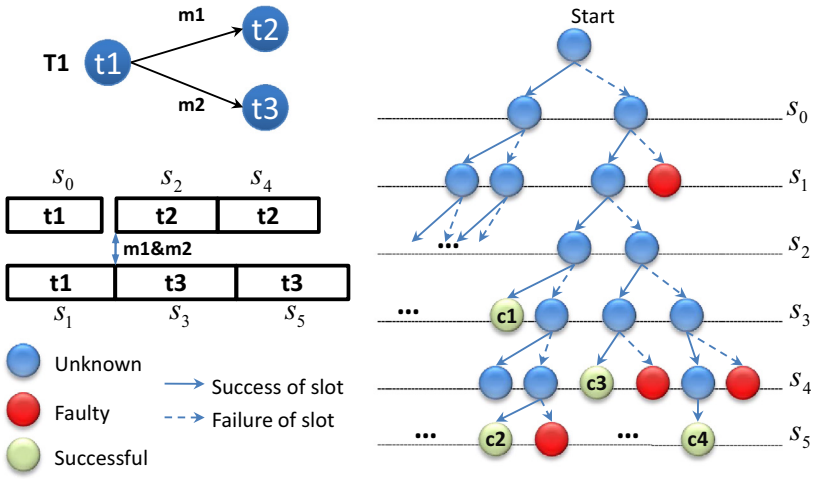

Fig. 5. Tree-based reliability analysis.

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

4.3. Designing fault-tolerant systems using imperfect fault detectors

This section presents an extension of the DSE approach to take embedded fault detection into account. As mentioned previously, existing approaches that consider embedded fault detection assume that all transient faults are detected at the end of the task execution. This assumption simplifies the problem but is problematic in practice. On the one hand, a perfect detector might not exist or is difficult to implement, making the algorithms developed under this assumption less useful. On the other hand, even if implementable, perfect detectors typically come with high resource and timing overheads. In recent work [10,31] it has been shown that the time needed for high-coverage fault detection may become much longer than the execution time of the task itself (*e.g.*, the timing overhead could be 400% using techniques proposed in [10]).

The problem can be considered from a different angle. In the fault-tolerance community, researchers have developed various fault detection techniques that achieve a certain fault detection coverage and with a particular overhead [10,31]. It is critical to select which fault detector to implement for each individual task. By making the assumption of perfect fault detection, a *biased design decision* is made, which selects the most expensive fault detector for every task. Other design alternatives with partial fault detectors are ignored. Moreover, since fault detection causes timing overhead, selecting better fault detectors reduces the opportunity for spatial/temporal replication. Clearly, a tradeoff analysis is needed to find the optimal setup.

Taking imperfect fault detection into account, the execution of a task instance may result in three scenarios: (1) it executes successfully, (2) a transient fault occurs and is detected, and (3) a transient fault occurs and is not detected. A detectable fault can be easily handled by making the component fail-silent to stop error propagation. On the contrary, an undetected fault leads to the case that a wrong output is delivered to the subsequent tasks without warning. If a subsequent task receives different inputs from the replicated tasks, it has at this point no knowledge about which of the inputs is correct. In practice, voting is typically used to handle this dilemma. Since faults are considered as rare events, the majority among the set of inputs is considered to be correct.

We assume a majority voting mechanism is implemented for each task that has replicas available. The voter generates an output if and only if a dominating result (*i.e.*, a majority) is found. The overall execution of a task, considering all its replicas, could again result in the following 3 scenarios: (1) the task executes successfully (*SUC*): it experiences no fault or only some faults that are later corrected by the voter; (2) Detected Unrecoverable Faults (*DUF*): the voter fails to find a dominating result and thus produces no output; and (3) Silent Data Corruption (*SDC*): multiple faults occur and the incorrect outputs mask the correct one. Both *DUF* and *SDC* are unwanted behavior that negatively influences the system reliability.

The tree-based approach has to be extended to consider imperfect fault detection. On the one hand, each node in the tree will now spawn 3 child branches, representing three possible results of a task. On the other hand, voting has to be supported in the analysis. The updated analysis is explained using the following example.

Consider task t_1 that features three copies in the scenario depicted in Fig. 7. The qualitative execution results of all the replicas are described by a fault scenario. The fault scenario contains one variable x_l for each replica l, which can have three values: x_l is 1 if the task executes correctly; x_l is 0 if it fails silently (a fault occurs and is detected) and x_l is -1 if it produces an incorrect output (*i.e.*, a fault occurs and is not detected). In the example, if the fault scenario is $\mathbf{x} = \{1, 1, -1\}$, the incorrect output of $t_{1,3}$ is masked and the overall result is *SUC*. In the scenario $\mathbf{x} = \{1, 0, 0\}$, both $t_{1,2}$ and $t_{1,3}$ produce no result, and only the output from $t_{1,1}$ will be used. Hence, the overall result is also *SUC*. In the scenario $\mathbf{x} = \{1, 0, -1\}$, a correct and an incorrect output are sent to the voter. The voter cannot identify the correct input since no majority is found. In this case, no output is generated and the overall result is *DUF*. In the last example scenario $\mathbf{x} = \{-1, 1, -1\}$, two incorrect outputs are sent to the voter. Note that the fault scenarios model only the qualitative result ($0, 1$, or -1), but the voting is performed based on the real value of the tasks' outputs. Hence, if two outputs are incorrect, two cases might happen: (1) the two incorrect outputs are equal and mask the single correct one, resulting in a *SDC*; (2) the two incorrect outputs are unequal and the voter does not see a dominating value, resulting in a *DUF*. To stay on the safe side, we have to assume the first case (*SDC*), because the probabilities of the two cases are very difficult to be quantified, even if possible.[2]

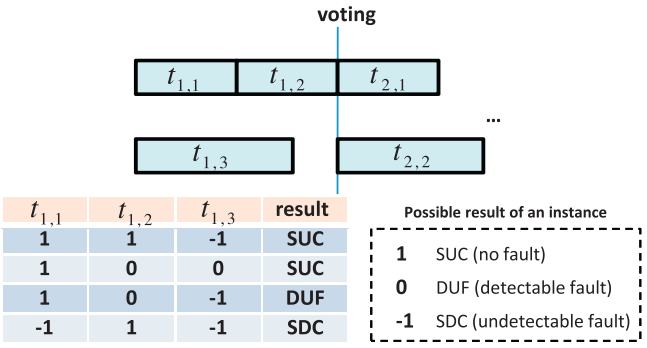

Fig. 7. Voting scenario analysis.

The analysis approach first performs a voting scenario analysis and computes the *SUC*, *DUF* and *SDC* probabilities for each individual task. Afterwards, these probabilities are combined to compute the system-level reliability as presented in [12]. It is important to note that the extended analysis has linear complexity in the number of tasks. The optimization procedure also needs adaptation to support imperfect fault detection. Here, the encoding scheme is extended to not only contain the mapping of tasks, but also an indication which fault detector is implemented by each task.

4.4. Code generation

After the design model has been updated and/or transformed by the DSE, the back-end of our framework speeds up the implementation by automatically generating (1) C source code of the application, and (2) platform configuration files required to execute the application.

4.4.1. Template-based code generation

A template-based transformation engine is used to generate text files from models. Templates are implemented in the *Xpand* language provided by EMF. They contain static parts that are used directly, as well as dynamic parts that are expanded based on the design model. To foster the modularity of the tool implementation, code templates adhere to an API (see below) and reside in the same Eclipse plugin as the corresponding meta-model elements.

4.4.2. Generation strategy

The automatic generation of implementation artifacts is driven by the updated system model resulting from the DSE.

[2] The probabilities are highly influenced by the application characteristics, the output data type, common cause errors, *etc.*

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

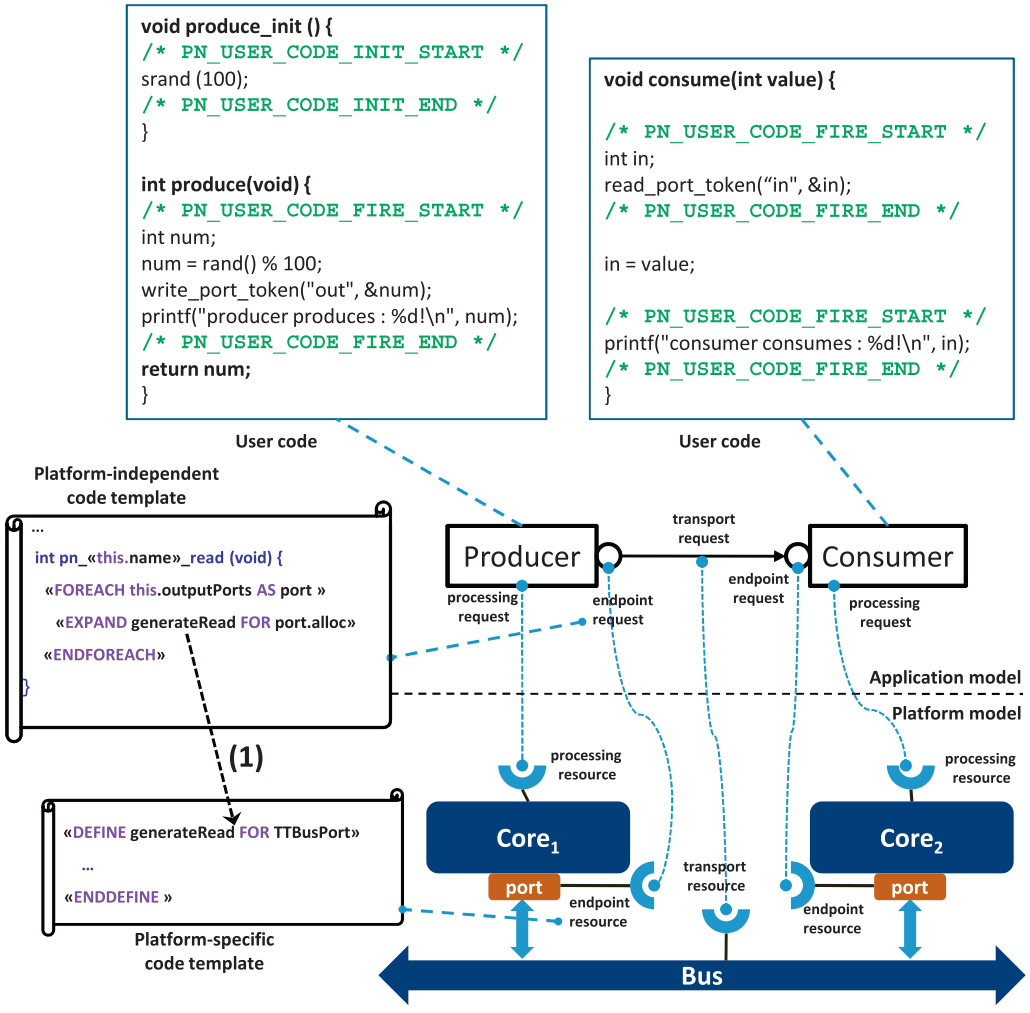

Fig. 8. Code generation approach.

First, the mapping of the application model to the platform model is analyzed in order to compute the set of deployment units required for the particular design (*e.g.*, OS executable or partition, bare-metal firmware image). Then, for each of the deployment units, the application code generation is performed by traversing the corresponding application model elements, tracing their mapping to the platform. Here, usually at least one source code file is generated for each the individual application components. Additionally, a main file is generated that implements the component initialization and the communication. In the following, we will illustrate the details using an example.

The build system required obtain binary images is generated automatically in form of CMake[3] scripts. As Eclipse is supported by CMake, both the system model and the source code can be developed within the same IDE. The generated CMake scripts consider the mapping of the application to heterogeneous deployment units (compiler tool-chain, *etc.*).

4.4.3. Example

During the discussion of the code generation we will distinguish between platform-independent (functional) and platform-specific (structural) code. Consider the simple KPN application in Fig. 8. Here, the producer generates a random integer which is printed to the console by the consumer. Our code generator creates one C source and header file for each KPN component. It contains the following subroutines (unique function names are ensured but ignored here to increase readability):

- `pn_init()/pn_done()`: (De-) initialization of the KPN component. It is called only once in the initialization (finalization) phase, before the execution of the KPN components starts (ends). Since KPN components are stateless [18], this code implements either technical aspects that are not covered by the model (*e.g.*, the initialization of the pseudo-random number generator for the *producer* component), or other structural code (*e.g.*, to open or close communication channels).
- `pn_read()`: This function executes a blocking read on all input ports of the KPN component and stores the data into a local buffer.
- `pn_fire()`: The computation kernel represented by the KPN component. It processes the input tokens and stores the results in a local output buffer.
- `pn_write()`: This function transfers the results from the local buffer to the output ports.

Additionally, the driver code that invokes these functions is generated. The template for the driver code traces the mapping of the processing request of the KPN components in order to delegate to a platform-specific template. In the example, both the producer and the consumer are directly mapped to a processor core. The corresponding processing resource is maintained in the

[3] http://www.cmake.org/.

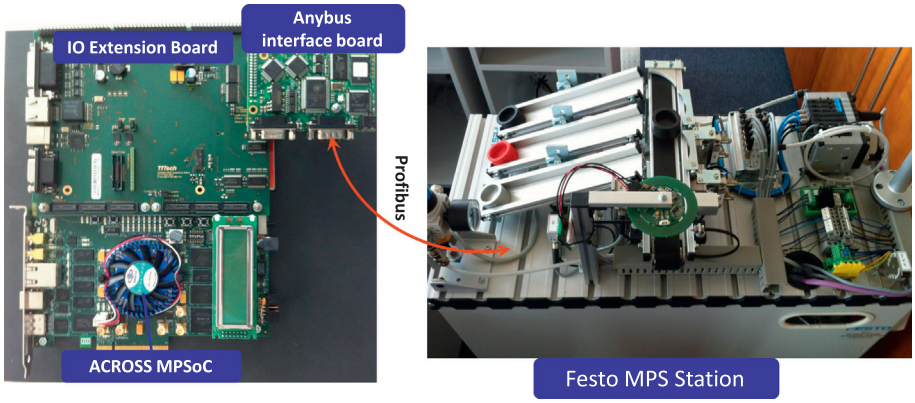

Fig. 9. Industrial control demonstrator.

`allocation` reference of class `Request` (see Fig. 4). The associated code template generates a `main()` function as driver code according to the execution model. In a time-triggered implementation, the `main()` function sequentially invokes `pn_read()`; `pn_fire()`; `pn_write()`; functions in the time slot allocated to the KPN component. In an event-triggered system, each KPN component is implemented in a separate execution context (thread, *etc.*). The blocking read guarantees that the execution adheres to the KPN semantics.

The `pn_fire()` function specifies the functional behavior of the component. Since the KPN model does not cover this aspect (black-box components), an additional behavioral specification is required in order to enable full code generation. Our framework provides two different alternatives (also used for `pn_init()`/ `pn_done()` if needed): a *fine-grained model* based on a meta-model with a pre-defined semantics can be used to fully model the behavior of a KPN component. If also the corresponding code templates are provided, functional code can be directly generated using a similar approach as previously presented for KPN. The IEC 61131-3 [14] meta-model included in our framework is an example for this approach that provides the behavioral specification in terms of data-flow and state-flow models.

If the enhanced analyzability provided by a formal application model is not required, or if legacy code needs to be integrated, KPN components can reference *(annotated) C source files*. Here, the sections to be extracted by the generator are marked up with dedicated C comments (see code fragments enclosed by `PN_USER_CODE_FIRE_START`, `PN_USER_CODE_FIRE_END`, *etc.*).

In contrast to that, the `pn_read()` and `pn_write()` functions contain purely structural code that maintains local communication buffers. Therefore, these functions can be derived automatically. Similar to the generation strategy for the processing resource (driver code), here the code templates for the KPN ports (communication endpoint request) delegate to platform-specific code templates associated to the corresponding end-point resource. For instance, code to send a message to $Core_2$ is generated for the output port of the *producer* (this step is marked with (1) in Fig. 8). In the user-supplied C code, an API is used to access the local buffers (`read_port_token()`, `write_port_token()`). The implementation of these functions is generated based on the information contained in the model. Here, the mapping of the KPN port to an end-point resource as well as its data type is considered. Therefore, the implementation details are encapsulated by generated auxiliary functions that ensure safe access to memory buffers. The generated communication code provides two layers of abstraction: `pn_read()` and `pn_write()` abstract the target platform and maintains local buffers. `read_port_token()` and `write_port_token()`) decouple the functional code from the actual buffer implementation.

As our modeling approach offers a flexible environment to allow modeling of a variety of platforms, the code generator is also designed in a modular way so that it can support different platforms. To achieve this goal, we strictly separate generic (functional) code and platform-dependent (structural) code in the code templates. The platform-dependent code is specified using dedicated code templates that are tightly coupled with the platform modeling objects. For example, each communication port type (*e.g.*, local memory buffer, partition-to-partition, core-to-core, *etc.*) is associated with specific templates that generate the code for reading/writing/initializing the port. Since the platform-specific code templates share a common interface, and *Xpand* supports polymorphism, the appropriate templates are transparently invoked by the code generator.

5. Case studies

We demonstrate the usability of our approach using two real-world applications. The first is an Adaptive Cruise Controller (ACC) from the automotive domain, the goal of which is to maintain the cruise speed while keeping safe distance from objects ahead. We execute this application using test cases automatically generated from the AutoFocus tool,[4] where simulated sensor values are stored as data arrays. The second application is a production line controller from the industrial automation domain. The control system has been tested in a real demonstrator setup built using the Modular Production System (MPS) from Festo Didactic[5] (see Fig. 9). It is used to control a conveyor-belt and several switches in order to sort incoming work-pieces. The MPS station is connected to the control system via Profibus [17].

The target architecture is the ACROSS MPSoC [6] running in an Altera Stratix IV FPGA. The MPSoC implements in total 8 Nios II soft-cores from Altera, 3 out of which are for general purpose application tasks. The application processors run the PikeOS operating system from SYSGO.[6] Communication between processors is realized using a Time-Triggered Network-on-Chip (TTNoC) architecture [28]. A dedicated I/O processor core is used as the gateway to off-chip resources, including I/O pins, sensors, actuators and network interfaces. A HSMC extension board provides the physical interfaces to the FPGA board hosting the MPSoC.

5.1. The ACC application

The ACC applications consists of 10 tasks communicating via 16 channels. We model the application structure using KPN as shown

[4] http://af3.fortiss.org/.
[5] http://www.festo-didactic.com.
[6] http://www.sysgo.com/.

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

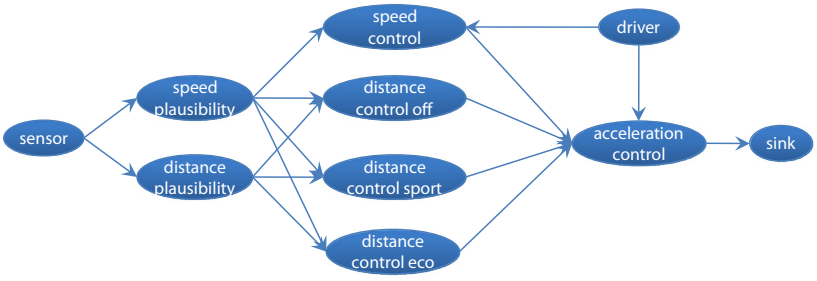

Fig. 10. Task graph of ACC.

in Fig. 10. The behavior of tasks is described using annotated C source files. In addition, a platform model is created to describe the ACROSS architecture. The following design exploration problem is considered for this application:

- *Design objectives:* the *reliability* of the application is to be *maximized*. The application is expected to have fail-operational behavior (in this case, a fault scenario that is only detectable but not correctable is considered as a failure). We use Failure In Time (FIT) as the metric of reliability, so the fitness value is to be minimized. At the same time, the *resource consumption* of the application is to be *minimized* (only processor time is considered).
- *Constraints:* An *end-to-end deadline* must be respected, *i.e.,* the maximum latency between the sensor and sink task must be smaller than the specified value. For a time-triggered design, the end-to-end latency can be directly obtained from the schedule. Additionally, *I/O constraints* must be fulfilled: The sensor, driver and sink tasks must be mapped to the I/O core (gateway to off-chip peripherals). The sensor task gathers information from the speed and distance sensors, the driver task collects the driver input and the sink task represents the actuator. Due to the unique availability of physical devices, these tasks cannot be replicated.

The MOEA optimizer is configured with a population of 100 and runs for 500 iterations. Since the two optimization objectives are conflicting with each other (higher reliability requires more redundancy and consumes more resources), the optimization result is not a single solution but a set of design alternatives (Pareto optimal solutions). The DSE tool generates a 2D figure that visualizes the tradeoff between these solutions in the objective space (see Fig. 11). The execution time of the DSE is around 120 s on a Windows machine with a 3 GHz CPU (single-thread).

The designer may select any of the solutions in the figure to look into the design details. Since we are considering a time-triggered system in this example, the design is represented as a Gantt chart, depicting the mapping and scheduling of all tasks. A screen shot is shown in Fig. 12. Here, dark blue time slots are used for application tasks, whereas light red time slots represent voting components. As it can be seen, the tasks are selectively replicated and distributed to the three application cores. In particular, a TMR scheme is implemented for the *DistanceControlEco* task using temporal replication, whereas a TMR scheme is implemented for the *AccelerationControl* task using spatial replication. Since *DistanceControlEco* generates input to *AccelerationControl*, a voter is inserted at each replica of *AccelerationControl*. The results of replicated *AccelerationControl* tasks are voted at the sink task.

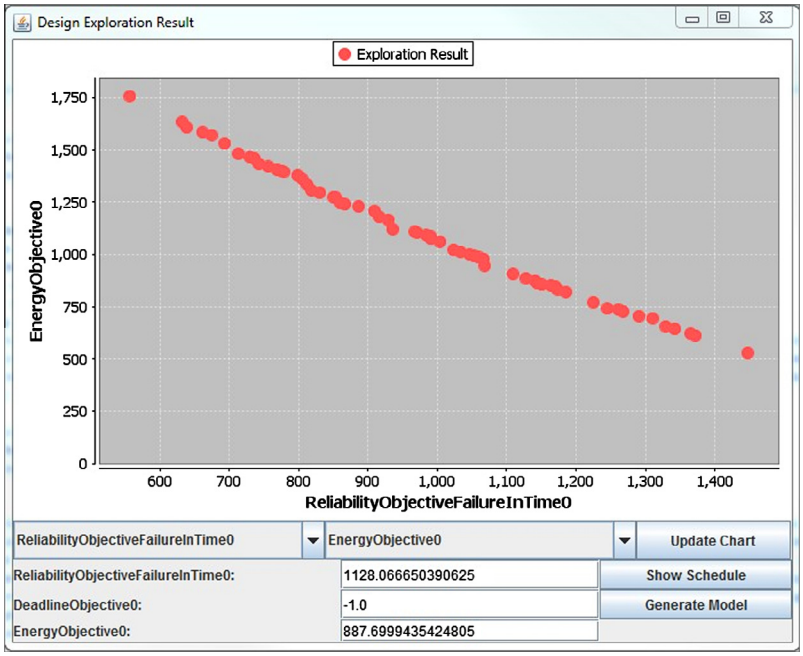

Fig. 11. DSE Tradeoff visualization (ACC study).

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

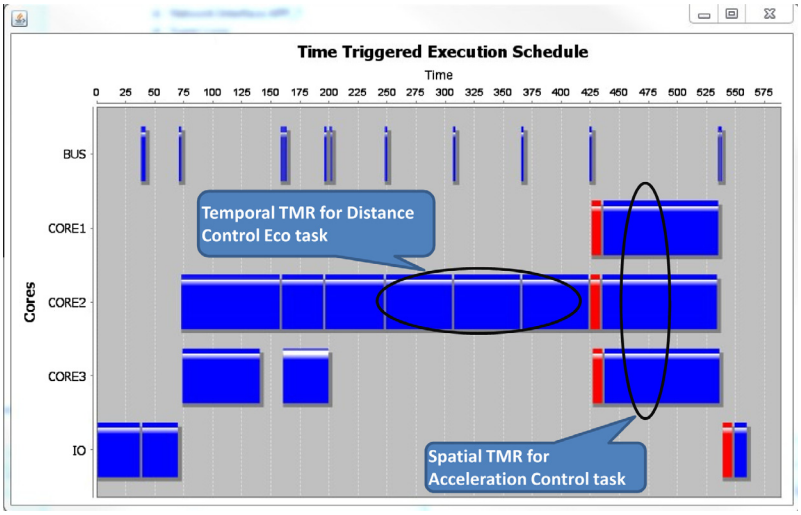

Fig. 12. Schedule visualization (ACC study).

Based on the tradeoff analysis of the DSE results, the designer selects the final implementation. An updated model is automatically generated using the *Generate Model* button from the GUI of our tool. The FTMs (replicated tasks, voting components, *etc.*) are explicitly instantiated in this updated model, so it can be directly used to synthesize the implementation (*i.e.,* source code for each core, required configuration files, *etc.*). The code generated for the ACC example has successfully been tested on the target system.

The modeling framework allows the designer to retarget the application to different hardware platforms with minor effort. The complete application model can be reused by replacing the platform model and re-running the DSE. For example, the generated ACC code has also been successfully deployed to a single Nios II processor running the PikeOS operating system. In this case, communication is performed via the PikeOS inter-partition-communication. Additionally, the design has been tested on the development host (Windows PC; communication via shared memory).

5.2. The production line controller

The development flow of this application is similar as for the ACC example. Again, KPN is used as a coarse-grained model to describe the application structure. On top of that, IEC 61131-3 SFC and FBD models are used to specify the behavior of the KPN components (sorting logic). Functional code can be automatically generated out of them. The DSE tool suggests a DMR implementation of the controller. Although the architecture provides three cores to implement TMR, it is not preferred here. This is because the application requires only fail-safe behavior (*i.e.,* stop the conveyor-belt), and while the TMR implementation achieves the same reliability as DMR, it consumes extra resources.

6. Conclusion

This paper tackles the problem of fault-tolerant embedded system design and presents an integrated framework that supports a complete reliability-aware design flow. Existing fault-tolerant scheduling approaches can be integrated to our framework to utilize the tool-supported (modeling) front-end and (code generation) back-end. In addition, we focus on removing unrealistic assumptions in current analysis and scheduling approaches to bring the research results more into practice. The ACC case study and the industrial control demonstrator illustrate the applicability of the proposed approach. As future work, we are interested in considering other non-functional properties in system design (*e.g., energy consumption) and investigate the reliability issues in mixed-criticality systems.*

References

[1] P. Axer, M. Sebastian, R. Ernst, Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints, in: CODES + ISSS, 2011, pp. 149–158.
[2] S. Barner, M. Geisinger, C. Buckl, A. Knoll, EasyLab: model-based development of software for mechatronic systems, in: IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications, Beijing, China, 2008, pp. 540–545.
[3] A. Benoit, L.C. Canon, E. Jeannot, Y. Robert, Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms, J. Sched. 15 (5) (2011) 615–627.
[4] C. Bolchini, A. Miele, Reliability-driven system-level synthesis for mixed-critical embedded systems, IEEE Trans. Comput. 62 (12) (2013) 2489–2502.
[5] S. Borkar, Designing reliable systems from unreliable components: the challenges of transistor variability and degradation, IEEE Micro 25 (6) (2005) 10–16.
[6] C. El Salloum, M. Elshuber, O. Höftberger, H. Isakovic, A. Wasicek, The ACROSS MPSoC – a new generation of multi-core processors designed for safety-critical embedded systems, in: DSD, 2012, pp. 105–113.
[7] P.H. Feiler, D.P. Gluch, J.J. Hudak, The architecture analysis & design language (AADL): an introduction. Tech. Note CMU/SEI-2006-TN-011, Carnegie Mellon University, 2006.
[8] A. Girault, H. Kalla, A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate, IEEE Trans. Dependable Secure Comput. 6 (4) (2009) 241–254.
[9] M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt, J. Teich, Reliability-aware system synthesis, in: DATE, 2007, pp. 409–414.
[10] G. Lyle, S. Chen, K. Pattabiraman, Z. Kalbarczyk, R. Iyer, An end-to-end approach for the automatic derivation of application-aware error detectors, in: DSN, 2009, pp. 584–589.
[11] J. Huang, J.O. Blech, A. Raabe, C. Buckl, A. Knoll, Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems, in: CODES + ISSS, 2011, pp. 247–256.
[12] J. Huang, K. Huang, A. Raabe, C. Buckl, A. Knoll, Towards fault-tolerant embedded systems with imperfect fault detection, in: DAC, 2012, pp. 188–196.
[13] J. Huang, A. Raabe, K. Huang, C. Buckl, A. Knoll, A framework for reliability-aware design exploration for MPSoC based systems, Des. Autom. Embed. Syst. 16 (4) (2013) 189–220.
[14] Int. Electrotechnical Commission: IEC 61131-3: Programmable controllers – Part 3: Programming Languages, 1993.
[15] V. Izosimov, I. Polian, P. Pop, P. Eles, Z. Peng, Analysis and optimization of fault-tolerant embedded systems with hardened processors, in: DATE, 2009, pp. 682–687.
[16] V. Izosimov, P. Pop, P. Eles, Z. Peng, Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems, in: DATE, vol. 2, 2005, pp. 864–869.
[17] W. Josef, K. Gerhard, Decentralization with PROFIBUS DP/DPV1, Publicis, 2003.
[18] G. Kahn, The semantics of simple language for parallel programming, in: IFIP Congress, 1974, pp. 471–475.
[19] N. Kandasamy, J. Hayes, B. Murray, Transparent recovery from intermittent faults in time-triggered distributed systems, IEEE Trans. Comput. 52 (2) (2003) 113–125.

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007

[20] C. LaFrieda, E. Ipek, J. Martinez, R. Manohar, Utilizing dynamically coupled cores to form a resilient chip multiprocessor, in: DSN, 2007, pp. 317–326.
[21] A. Lifa, P. Eles, Z. Peng, V. Izosimov, Hardware/software optimization of error detection implementation for real-time embedded systems, in: CODES + ISSS, 2010, pp. 41–50.
[22] M. Lukasiewycz, M. Glaß, C. Haubelt, J. Teich, SAT-decoding in evolutionary algorithms for discrete constrained optimization problems, in: CEC, 2007, pp. 935–942.
[23] R. Obermaisser, O. Höftberger, Fault containment in a reconfigurable multiprocessor system-on-a-chip, in: International Symposium on Industrial Electronics (ISIE), 2011, pp. 1561–1568.
[24] OMG: Systems Modeling Language 1.2, 2010.
[25] OMG: UML profile for MARTE 1.1, 2011.
[26] R. Passerone, I.B. Hafaiedh, S. Graf, A. Benveniste, D. Cancila, A. Cuccuru, S. Gerard, F. Terrier, W. Damm, A. Ferrari, L. Mangeruca, B. Josko, T. Peikenkamp, A. Sangiovanni-Vincentelli, Metamodels in Europe: languages, tools, and applications, IEEE Des. Test. Comput. 26 (3) (2009) 38–53.
[27] K. Pattabiraman, Z. Kalbarczyk, R. Iyer, Automated derivation of application-aware error detectors using static analysis: the trusted Illiac approach, IEEE Trans. Dependable Secure Comput. 8 (1) (2011) 44–57.
[28] C. Paukovits, H. Kopetz, Concepts of switching in the time-triggered network-on-chip, in: RTCSA, 2008, pp.120–129.
[29] P. Pop, V. Izosimov, P. Eles, Z. Peng, Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication, IEEE Trans. VLSI Syst. 17 (3) (2009) 389–402.
[30] P.K. Saraswat, P. Pop, J. Madsen, Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems, in: RTAS, 2010, pp. 89–98.
[31] U. Schiffel, A. Schmitt, M. Süßkraut, C. Fetzer, Software-implemented hardware error detection: costs and gains, in: DEPEND, 2010, pp. 51–57.
[32] S.M. Shatz, J.P. Wang, Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems, IEEE Trans. Rel. 38 (1) (1989) 16–27.
[33] P.V. Stralen, A. Pimentel, A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs, in: CODES + ISSS, 2012, pp. 393–402.
[34] L. Thiele, I. Bacivarov, W. Haid, K. Huang, Mapping applications to tiled multiprocessor embedded systems, in: ACSD, 2007, pp. 29–40.
[35] Y. Xiang, T. Chantem, R.P. Dick, X.S. Hu, L. Shang, System-level reliability modeling for MPSoCs, in: CODES + ISSS, 2010, pp. 297–306.
[36] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, M. Irwin, Reliability-aware co-synthesis for embedded systems, in: ASAP, 2004, pp. 41–50.
[37] D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks, IEEE Trans. Comput. 58 (10) (2009) 1382–1397.

Jia Huang received his master degree in Communications Engineering from RWTH Aachen, Germany in 2009. He is currently a research assistant at fortiss Institute and a PhD candidate at Technical University of Munich. His main research interests are embedded software and Hardware–Software Codesign for embedded and real-time systems.

Simon Barner received his diploma in Informatics at Technische Universität München in 2006. Since 2011 he is working as a doctoral candidate at fortiss. His research interests focus on modeling of embedded systems as well as automatic generation of code and platform configuration.

Andreas Raabe received his doctorate in 2008 in the field of Hardware–Software-Codesign. Subsequent to a postdoctoral fellowship with the International Computer Science Institute Berkeley, he is heading the "Parallel Architecture" group at fortiss. His research interests include Hardware–Software-Codesign, Computer Architecture, and Platform-dependent Software-design.

Christian Buckl received his PhD in Informatics from the Technische Unversität München in 2008. Since 2009 he is leading the reserch department "Cyber-Physical Systems" at fortiss. The research focus of his department is on model-driven development with a focus on extra-functional requirements, software architectures for Cyber-Physical Systems, computer aided synthesis and verification, and virtual engineering and robotics. The research results are applied prototypically in the domains of automotive and automation.

Alois C. Knoll received the diploma (M.Sc.) degree in Electrical/Communications Engineering from the University of Stuttgart, Germany, in 1985 and his PhD (summa cum laude) in computer science from the Technical University of Berlin, Germany, in 1988. He served on the faculty of the computer science department of TU Berlin until 1993, when he qualified for teaching computer science at a university (habilitation). He then joined the Technical Faculty of the University of Bielefeld, where he was a full professor and the director of the research group Technical Informatics until 2001. Between May 2001 and April 2004 he was a member of the board of directors of the Fraunhofer-Institute for Autonomous Intelligent Systems. At AIS he was head of the research group "Robotics Construction Kits", dedicated to research and development in the area of educational robotics. Since autumn 2001 he has been a professor of Computer Science at the Computer Science Department of the Technische Universität München. He is also on the board of directors of the Central Institute of Medical Technology at TUM (IMETUM-Garching); between April 2004 and March 2006 he was Executive Director of the Institute of Computer Science at TUM.
His research interests include cognitive, medical and sensor-based robotics, multi-agent systems, data fusion, adaptive systems and multimedia information retrieval. In these fields he has published over 200 technical papers and guest-edited international journals. He has participated (and has coordinated) several large scale national collaborative research projects (funded by the EU, the DFG, the DAAD, the state of North-Rhine-Westphalia). He initiated and was the program chairman of the First IEEE/RAS Conference on Humanoid Robots (IEEE-RAS/RSJ Humanoids2000), he was general chair of IEEE Humanoids2003 and general chair of Robotik 2004, the largest German conference on robotics, and he served on several other organizing committees. Prof. Knoll is a member of the German Society for Computer Science (Gesellschaft für Informatik (GI)) and the IEEE.

Please cite this article in press as: J. Huang et al., A framework for reliability-aware embedded system design on multiprocessor platforms, Microprocess. Microsyst. (2014), http://dx.doi.org/10.1016/j.micpro.2014.02.007