Integrate Online Model Checking into Distributed Reconfigurable System on Chip with Adaptable OS Services

Sufyan Samara, Yuhong Zhao*, and Franz J. Rammig

Heinz Nixdorf Institute, University of Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
sufyan@mail.uni-paderborn.de

Abstract. This paper presents a novel flexible, dependable, and reliable operating system design for distributed reconfigurable system on chip. The dependability and reliability are achieved by integrating online model checking technique. Each OS service has different implementations which are further partitioned into small blocks. This operating system design allows the OS service to be adapted at runtime according to the given resource requirements and response time. Such adaptable services may be required by real time safety-critical applications. The flexibility introduced in executing adaptable OS services also gives rise to a potential safety problem. Thus, online model checking is integrated to the operating system so as to improve the dependability, reliability, and fault tolerance of these adaptable OS services.

Keywords: Online Model Checking, Distributed Reconfigurable System on Chip, OS Adaptable Service.

1 Introduction

The vast growing need for powerful yet small and customized systems encouraged the development of embedded systems. These now are most likely to contain more than one computational element on a single chip forming what so called a System on Chip (SoC). An addition of a Field Programmable Gate Array (FPGA) gives SoC the ability of reconfiguration. FPGAs are known of their computational power and dynamic behavior in comparison with General Purpose Processors (GPP). Many applications such as signal processing, encryption/decryption, and multimedia encoding/decoding are in need for such systems. However, the complexity of these systems is no longer easily manageable, especially if they are distributed. This raises the necessity for embedded Operating System (OS).

* This work is developed in the course of the Collaborative Research Center 614 - Self-Optimizing Concepts and Structures in Mechanical Engineering - Paderborn University, and is published on its behalf and funded by the Deutsche Forschungsgemeinschaft (DFG).

M. Hinchey et al. (Eds.): DIPES/BICC 2010, IFIP AICT 329, pp. 102–113, 2010.

An OS working on distributed Reconfigurable SoCs (RSoC) can also benefit from the dynamic behavior and the computational power provided by their FPGAs. These benefits involve adaptability, runtime safety checking and recovery, etc. This adaptable OS is capable of changing its resource requirements and execution behavior in order to accommodate the variety of applications highly expected in distributed systems.

The underlying OS services usually play a critical role in order to safely execute real time applications/tasks on distributed RSoCs. This is reflected in ensuring the executions of these services *correct* without violating any deadlines or safety constraints. In order to achieve this goal, we make OS services be accompanied with a runtime checker to predict possible errors or constraints violation. In case that an error or constraint violation is found, the presented novel OS service design allows the service to recover. The recovery process allows the OS service to continue execution from the point the error occurred with as minimum losses as possible and without violating any constraints. This is achieved by recalculating and finding another configuration efficiently [1,2].

In this paper we present a novel flexible and dependable OS design that can adapt at runtime its services according to applications/tasks desired QoS, i.e., the actual resource requirements and response time, on the one hand; meanwhile online check at model level the safety constraints of these adaptable services and then recover if necessary from the detected errors or constraints violation on the other hand. We make this design feasible by partitioning OS services and integrating online model checking [3,4] into the operating system.

The rest of this paper is organized as follows: Section 2 introduces the distributed system topology and discusses the adaptable OS design; in Section 3 the online model checking is introduced as follows: Subsection 3.1 explains application scenario; Subsection 3.2 describes how to generate an abstract model for an adaptable OS service; the model checking paradigm and the pre and post- checking are presented in Subsection 3.3 and 3.4 respectively; the integration of the online model checker with the OS design is discussed in Section 4; in Section 5 some related work is introduced and finally we conclude the work in Section 6.

2 Distributed Reconfigurable System on Chip with Adaptable OS Services

The distributed system under consideration is a *hybrid* one between a centralized and fully distributed system. The RSoCs are distributed and allowed to operate and communicate freely without central coordination. However, at the initialization stages and for the sake of OS services distribution, a unique central RSoC exists. This central RSoC is assumed to have enough resources to hold and manage a whole copy of the OS and acts as OS Services Repository (OSR), see Figure 1. After distributing the services, this central RSoC existence is no more important, but beneficent. This is because the used distribution algorithm supports encoding which allows the retrieval of any OS services and provides for some fault tolerance.

Each RSoC in the distributed system has at least an FPGA and GPP. For an OS service to be able to fully or partially utilize FPGA and GPP on each

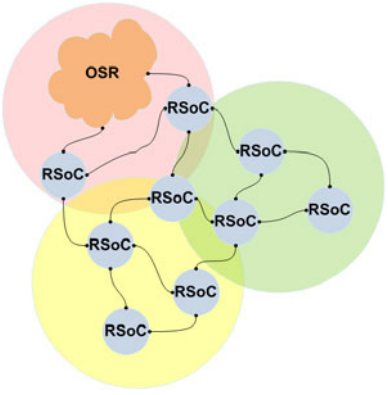

Fig. 1. RSoCs distribution

RSoC, the service exists in two implementations. One implementation is to run on FPGA while the other one is to run on GPP. Further, each of the two implementations is divided into the same number of blocks, where each block is called a Small Execution Segment (SES), see Figure 2. Each SES with the same index has the same functional behavior. For instance, the second SES in all the implementations of the given OS service will have the same functional behavior. If we input identical data to the second SES in any implementation of this OS service, we should get identical output. However, the time of getting these outputs, the power consumed, and the area/utilization required for each SES differ.

These similarities and differences give us much flexibility in executing an OS service. For example, if we have one service with two implementations each with just two SESs, we would have four possibilities to execute the service, each with different resource requirements and response time. Our previous work, [1] and [2], prove the feasibility of this design to execute and adapt an OS service to run on RSoC even with very limited resources without violating any demanded constraints. Further, a complete formal description of the design and a linear algorithm to schedule such service were presented.

As aforesaid, the RSoCs are distributed in a hybrid topology network where a central RSoC management node exists but the RSoCs can work without central RSOC. This central RSoC contains the whole set of OS services. It is used in the initialization stage to balance the distribution of the services/SESs over all the available RSoCs in the system. Depending on resources and distribution, an execution of a service can be either carried out on a single RSoC or as collaboration of more than one RSoC. This requires an evaluation at runtime to find a suitable configuration for the service to execute on available resources without violating any constraint. Due to dynamic changes in applications/tasks, the resources or the constraints may change accordingly at runtime. This may lead to change/adjustment in service configuration. Due to the sensitivity of the process, as this may be a service requested by a real time application/task, dependability and fault tolerance of the underlying operating system is highly expected.

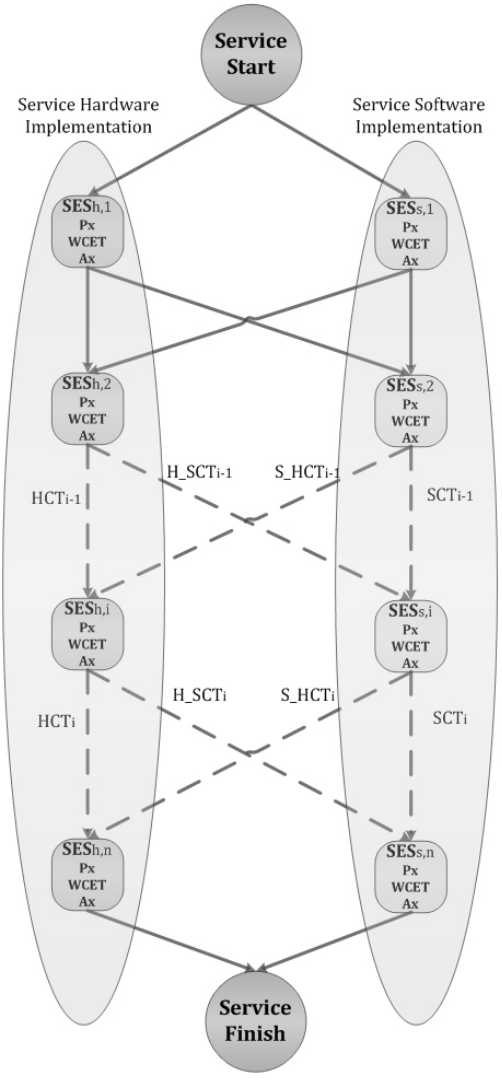

Fig. 2. Two implementations of OS service with execution order of the SES's. Here S stands Software, H: for Hardware, C for Communication, T for Time, P for power and A for Area. For example SCT denotes Software Communication Time.

3 Online Model Checking

3.1 Application Scenario

Given an adaptable service with n (> 1) implementations and each implementation with m (> 1) SES's, then we have m^n different combination ways to configure the SES's of this OS service. The actual configuration of the SES's has

to be determined at runtime with respect to the resource requirements and the response time. There is no way to know in advance which configuration of the SES's should be selected. In essence, each configuration of the SES's can be seen as a different implementation of this service. Due to the huge time and space complexity, it is impossible to verify the safety properties of all the m^n different implementations of this service at the systems development phase. Therefore, online model checking [3,4] might be a good choice to improve the dependability and fault tolerance of our distributed RSoC with adaptable OS services.

Without loss of generality, let's suppose that some SES's of a given OS service are safety-critical. Of course, all the SES's of an OS service can be conservatively labeled as safety-critical. When this service is called by a real-time application/task running on an RSoC, the middleware (see Section 4) of the RSoC will figure out a suitable configuration of this service. Therefore, the middleware always knows in advance when a safety-critical SES will be executed and thus can trigger online model checking in time on an RSoC with enough resource.

3.2 Abstract Model

In order to do online model checking, we need to generate a sufficiently precise abstract model from the source code of each safety-critical SES of a service at first. A promising approach to construct an abstract state graph automatically is *predicate abstraction* [5]. Let $\{\varphi_1, \varphi_2, \cdots, \varphi_k\}$ $(k > 0)$ be a set of predicates induced from the conditional statements and guarded expressions of the source code. E.g., for a guard $(x < y)$, where x and y are Integer variables, we can get a predicate $\varphi = (x < y)$. Abstract states are the evaluations of these predicates $\varphi_1, \varphi_2, \cdots, \varphi_k$ on the program variables at each statement of the source code.

The abstract state graph is constructed starting from the abstract initial state. With the help of some theorem prover (e.g., PVS), we can compute the possible successors of any abstract state by deciding for each index i whether φ_i or $\neg\varphi_i$ is a *post* condition of this abstract state. Obviously, the more predicates we have, the more precise the abstract model is. The resulting abstract model is an over approximation of the concrete system. For every concrete state sequence, there exists a corresponding abstract state sequence.

The relations between concrete states and abstract states are defined by means of two functions: *abstraction* function α maps every set of concrete states to a corresponding abstract state; *concretization* function γ maps every abstract state to a set of concrete states that it represents.

In this way, for each safety-critical SES_i of a service, we can get the corresponding abstract model $\widehat{SES_i}$ as well as the abstraction function α_i and concretization function γ_i.

3.3 Model Checking Paradigm

Online model checking runs on an RSoC in parallel with the SES's to be checked. The abstract model of the checked SES is explored with respect to the given

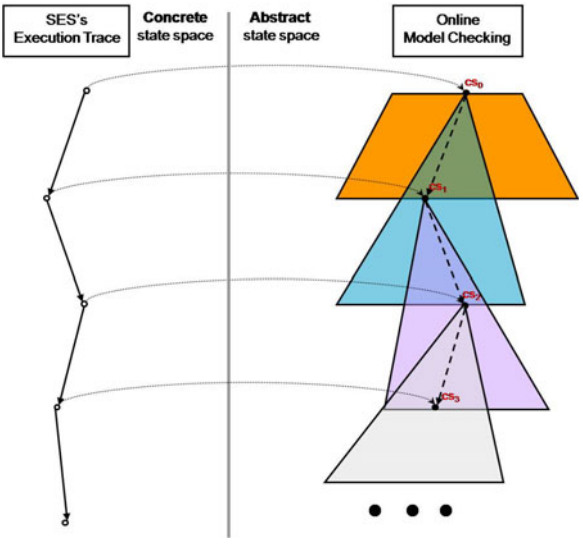

Fig. 3. Online Model Checking

safety property. Here we consider such a kind of the safety properties that can be formally specified as invariants or LTL formulas. Since our model checking is done online, the current (concrete) states of the SES's execution can be monitored and reported to the model checker from time to time (see Subsection 4.1). By mapping these concrete states to the corresponding abstract states at model level through the abstraction function, the online model checker needs only to explore such a partial state space reachable from these abstract (current) states as shown in Fig 3. Initially, when the current starting state is not available, online model checking will explore all the possible behaviors from the abstract starting state. Once a current state is available, the state space to be checked will be shrunk to the part reachable from the corresponding abstract state. If this partial abstract model is checked *safe* against the given property, then we have more confidence to the safety of the actual execution trace. It doesn't matter even though there might be some errors lurked outside the partial abstract state space. If an error is detected within the partial state space, the recovery process will be triggered (see Subsection 4.2). Notice that the detected error might not really exist in the source code, because we check at (abstract) model level, which usually contains more behaviors than the corresponding source code. To avoid the error really to happen, it is necessary for safety-critical systems to trigger recovery process.

We have done some experiments to estimate the performance of our online model checking for invariants and LTL formulas [4]. The experimental results are promising and demonstrate that the maximal out-degree of a model has a larger influence on look-ahead performance than the average degree of the model.

3.4 Pre-checking and Post-checking

Since we check the abstract model instead of the concrete source code, the execution of our online model checker is loosely bound to the execution of the source code. Online model checking is able to explore the abstract state space even when a current state is not yet monitored. In fact, the current states are only used to reduce the state space to be checked. From this point of view, there exists a race between the model checker and the source code to be checked.

Ideally, we wish that model checker could always run enough (time) steps ahead the execution of the source code. This depends on the complexity of the checking task as well as the underlying hardware architecture. In reality, model checker might fall behind the execution of the source code. Therefore, we introduce two checking modes: *pre-checking* and *post-checking*. The model checking is in pre-checking mode, if it runs ahead of the execution of the source code; otherwise, it is in post-checking mode.

In pre-checking mode, the model checker can predict violations before they really happen. In post-checking mode, it seems at first sight that the violations could only be detected after they have already happened. However, it is still possible to "predict" violations even in post-checking mode because our on-line checking works at the model level. If an error is detected at some place other than the monitored execution trace in the partial state space, then we can "predict" that there might be an error in the model which has not happened yet. In this sense, both checking modes are useful for safety-critical systems.

Of course, we hope that the model checker can take the leading position against the source code for as long time as possible. We need to find a sophisticated strategy to make the model checking have more chance or higher probability to *win* against the source code. Recall that the source code is usually validated by means of simulation and testing. Currently we are looking to find some heuristic knowledge at the system testing phase so that the abstract model can be enriched with more useful information. The heuristic information can thus guide on-line model checker to further reduce the state space to be explored whenever necessary.

4 Integrate Online Model Checking into Distributed RSoC with Adaptable OS Services

Recall that every RSoC in the distributed system is assumed to have at least one FPGA and one GPP. In order to hide the complexity and provide some transparency to applications, a middleware is introduced to each RSoC. The middleware can prepare the services needed by applications, coordinate the communication between applications and OS services, monitor the resources availability, and manages the SES's. It can also cooperate with the online Model Checker (MC), see Figure 4-A, to provide for fault tolerance and recovery as discussed bellow.

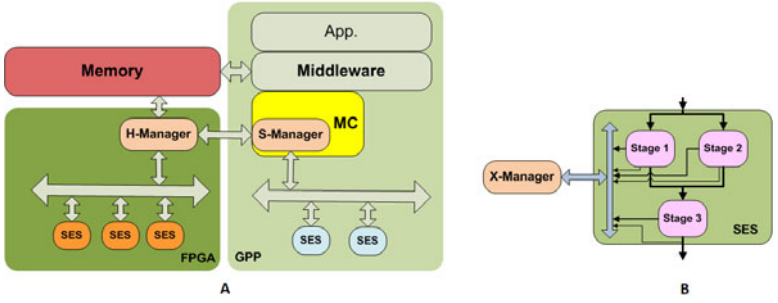

Fig. 4. A) An overview of an RSoC with one FPGA and GPP and the communications between SESs, MC, and memory. B) An insight of the SES stages and the communication with the X-manager.

4.1 Communication between the MC and SES's

Online model checker needs to communicate with SES's running on FPGA as well as on GPP. For the implementations of an adaptable service, all the SES's with the same index have the same functional behavior, i.e., given the same input, we will get the same output from the SES's with the same index. However, the SES's with the same index might be implemented by different algorithms in different languages on different platforms. They might be executed in a parallel or sequential way depending on the underlying architecture. As a result, the SES's with the same index are distinctive in terms of execution time and resource usage. This makes it unrealistic to synchronously schedule the communication between the model checker and an SES to be checked, e.g., every T time. Fortunately, an event driven approach is a suitable solution. This can be done at the design phase by defining triggering points at which communication is initiated. The triggering points depends on the functional behavior and states of each OS service and can be obtained by analyzing the source code. When the communication is triggered, a decoded message with the required information is sent to the model checker. These are normally global data and conditional values.

The model checker is usually running on GPP. This eases data transferring between SES's running on GPP and the model checker. It can be achieved by coping/accessing the address space of the SES or using shared memory. Because we also consider to check the SES's running on FPGA, an X-manager is introduced, where X is either H for hardware or S for software. The X-manager is working as complementary part to the model checker. It consists of two parts: the H-manager which works on FPGA and the S-manager which runs with the model checker on GPP, see Figure 4-A. All the SES's communications involving reading or writing memory are done through the X-manager. In doing so, the X-manager can monitor all the modifications and synchronize the SES's to get an updated value to any memory request. Without the X-manager the memory requested by SES may not be up to date, because FPGA can normally access the physical memory directly. In addition, anything running on GPP accesses

memory using a memory manager. This may also involve caching and/or virtual memory usage. In this case, memory accessed by an SES on FPGA might not be up to date if that memory was cached by an SES on GPP, and no update occurs to the physical original memory. To avoid such errors, we need to monitor and synchronize any modification to memory so as to ensure that we always read up to date values.

Needless to say, X-manager plays a decisive role in the communication between the online model checker and the SES to be checked. For this purpose, the internal structure of each SES is further defined into stages at design phase as shown in Figure 4-B. A stage is a logical grouping of a SES code after which the MC is triggered. Any access to the memory from every stage is monitored by the X-manager. At the end of each stage, an event will be sent to the X-manager. On receiving the event, the X-manager will provide the model checker with a snap shot of the memory just modified by the so far executed stage of the SES to be checked. This minimizes the time needed to transfer data to the model checker. Thus, the model checker might have more chance to run in pre-checking mode, i.e., look ahead in the near future at the model level. This is important as it allows the recovery process to happen without violating any constraints.

4.2 Recovery Process

When an application/task running on an RSoC requests a service to be executed, the middleware of the RSoC evaluates the available resources and the real time demands of the application/task to find a suitable configuration. This process may require coordination with other RSoCs or if applicable with an Operating Systems Repository (OSR) (see Fig. 1). It is just at this time that the configuration of the service to be executed is known. This configuration is then administrated by the the middleware of the RSoC for execution. The online model checker is triggered whenever an SES marked with safety-critical is known to be executed. The model checker runs in parallel with the SES to be checked. In case that a possible violation is detected in the abstract model of the SES to be checked, a recovery process is initiated as shown in Fig 5.

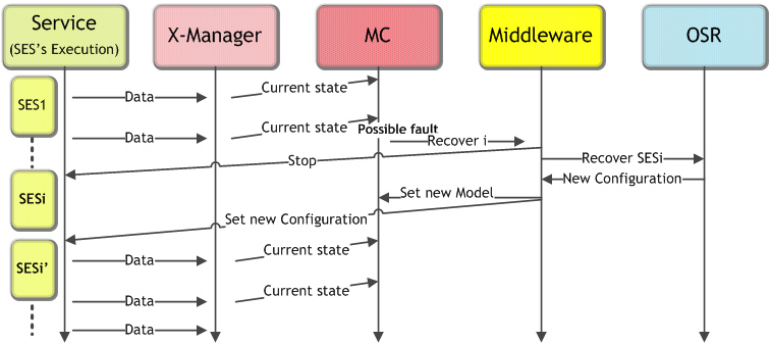

Fig. 5. Recovering a possible fault in OS service execution

The recovery process is initiated by sending a recovery message from the model checker to the RSoC middleware which coordinates the process of OS service execution. This can be the same RSoC where the model checker is being executed or different RSoC in the distributed system. The message contains information about the possible incorrect SES and the current state of this SES. The middleware then evaluates a new suitable configuration and if necessary acquires any missing SES from OSR. Meanwhile, it raises a stop signal to the incorrect service. The middleware afterwards initialize the new configuration and redirect the further execution of the service to it. The middleware also sends the model checker a message to set a new model associated with the new SES to be executed if necessary.

5 Related Work

The benefits of implementing OS partially on RSoC is well recognized [6,7]. Many researches have been carried out in the direction for an OS design to support RSoC [8,9,10]. The idea of multiple implementations was also presented in [11] and [12]. Every Service were assumed to have one implementation as a whole for GPP and another for FPGA. Both implementations co-exist at the same time on an RSoC to allow context switching. Doing so wastes too much resources on each RSoC and makes the underlying OS design unfeasible for distributed systems. Moreover, this OS design uses an algorithm based on Binary Integer Programming (BIP), whose computational complexity is $O(n^2)$. We present an OS design specially for distributed systems. It is fault tolerant and it can adapt according to the QoS requirements of the applications. Further, the adopted algorithm [1,2] has a linear complexity.

For distributed systems, correctness and temporal analysis of the underlying OS are desirable, because most SoC systems have real-time and dependability requirements [13]. This analysis includes more and more formal verification techniques like model checking [14]. Model checking has the advantage of being fully automated and inherently includes means for diagnosis in case of errors. On the other hand, model checking is substantially confronted with the so called state explosion problem. Numerous approaches to overcome this deficiency have been developed, like partial order reduction [15], compositional reasoning [16], and other simplification and abstraction techniques, which aim to reduce the state space to be explored by *over-approximation* [17] or *under-approximation* [18] techniques.

In recent years, runtime verification is presented as a complementary approach to the static checking techniques. The basic idea of the state-of-the-art runtime verification [19,20,21,22] is to monitor the execution of the source code and afterwards to check the so far observed execution trace against the given properties specified usually by LTL formulas. The checking progress always falls behind the execution of the source code because the checking procedure can continue only after a new state has been observed. In contrast, our runtime verification is applied to the model level. The states observed from the execution trace are

mainly used to reduce the state space to be explored at the model level. If the checking speed is fast enough, our online model checking could keep looking certain time steps ahead of the execution of the source code and then predict how many time steps in the near future are safe.

6 Conclusion

We present a novel OS design which allows OS services to have flexible QoS, fault tolerance, and recovery ability in distributed RSoC environment. Each OS service may have different implementations, which can be configured at runtime. This dynamic feature makes it difficult to check the safety of such OS services at design phase. Therefore, online model checker is integrated so as to make error prediction and recovery available. In [1] and [2], we have proved the feasibility of this OS design to execute and adapt an OS service running on RSoC even with very limited resources without violating any demanded requirements. We have done some experiments to estimate the performance of our online model checking for invariants and LTL formulas [4]. The experimental results are promising and demonstrate that the maximal out-degree of a model has a larger influence on look-ahead performance than the average degree of the model.

References

1. Samara, S., Tariq, F.B., Kerstan, T., Stahl, K.: Applications adaptable execution path for operating system services on a distributed reconfigurable system on chip. In: ICESS 2009: Proceedings of the 2009 International Conference on Embedded Software and Systems, Washington, DC, USA, pp. 461–466. IEEE Computer Society, Washington (2009)
2. Samara, S., Schomaker, G.: Self-adaptive os service model in relaxed resource distributed reconfigurable system on chip (rsoc). In: Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Washington, DC, USA, pp. 1–8. IEEE Computer Society, Washington (2009)
3. Zhao, Y., Rammig, F.J.: Model-based runtime verification framework. Electronic Notes in Theoretical Computer Science 253(1), 179–193 (2009)
4. Rammig, F.J., Zhao, Y., Samara, S.: On-line model checking as operating system service. In: Lee, S., Narasimhan, P. (eds.) SEUS 2009. LNCS, vol. 5860, pp. 131–143. Springer, Heidelberg (2009)
5. Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
6. Engel, F., Kuz, I., Petters, S.M., Ruocco, S.: Operating systems on socs: A good idea? National ICT Australia Ltd. (2004)
7. Wigley, G., Kearney, D.: Research issues in operating systems for reconfigurable computing. In: Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms (ERSA), pp. 10–16. CSREA Press (2002)
8. Donato, A., Ferrandi, F., Santamberogio, M., Sciuto, D.: Operating system support for dynamically reconfigurable soc. architectures. Politecnico di Milano (2006)

9. Nollet, V., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Designing an operating system for a heterogeneous reconfigurable soc. In: Proceedings of the RAW 2003 workshop (2003)
10. Walder, H., Platzner, M.: A runtime environment for reconfigurable hardware operating systems. In: Becker, J., Platzner, M., Vernalde, S. (eds.) FPL 2004. LNCS, vol. 3203, pp. 831–835. Springer, Heidelberg (2004)
11. Götz, M., Rettberg, A., Pereira, C.E.: Run-time reconfigurable real-time operating system for hybrid execution platforms. In: Information Control Problems in Manufacturing, IFAC (2006)
12. Götz, M., Rettberg, A., Pereira, C.E.: Towards run-time partitioning of a real time operating system for reconfigurable systems on chip. In: IFIP International Federation for Information Processing (2005)
13. Gajski, D.D., Vahid, F.: Specification and design of embedded hardware-software systems. IEEE, Design & Test of Computers 12, 53–67 (1995)
14. Clark, E.M., Grumberg Jr., O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
15. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem, Secaucus, NJ, USA. Springer, New York (1996), Foreword By-Wolper, Pierre
16. Berezin, S., Campos, S.V.A., Clarke, E.M.: Compositional reasoning in model checking. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 81–102. Springer, Heidelberg (1998)
17. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)
18. Lee, W., Pardo, A., Jang, J.Y., Hachtel, G., Somenzi, F.: Tearing based automatic abstraction for ctl model checking. In: ICCAD 1996: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, pp. 76–81. IEEE Computer Society, Los Alamitos (1996)
19. Barnett, M., Schulte, W.: Spying on components: A runtime verification technique. In: Leavens, G.T., Sitaraman, M., Giannakopoulou, D., eds.: Workshop on Specification and Verification of Component-Based Systems. October 2001, Published as Iowa State Technical Report 01-09a (October 2001)
20. Arkoudas, K., Rinard, M.: Deductive Runtime Certification. In: Proceedings of the 2004 Workshop on Runtime Verification (RV 2004), Barcelona, Spain (April 2004)
21. Chen, F., Rosu, G.: Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In: Proceedings of the 2003 Workshop on Runtime Verification (RV 2003), Boulder, Colorado, USA (2003)
22. Havelund, K., Rosu, G.: Java PathExplorer — a runtime verification tool. In: Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS 2001), Montreal, Canada (June 2001)