Software Composability and Mixed Criticality for Triple Modular Redundant Architectures

Stefan Resch, Andreas Steininger, Christoph Scherrer

Software Composability and Mixed Criticality for Triple Modular Redundant Architectures

Stefan Resch[1]**, Andreas Steininger[2], and Christoph Scherrer[1]**

[1] Thales Austria GmbH,
Handelskai 92, A-1200 Vienna
{stefan.resch, christoph.scherrer}@thalesgroup.com
[2] Vienna University of Technology, Embedded Computing Systems Group E182-2,
Treitlstr. 3, A-1040 Vienna,
steininger@ecs.tuwien.ac.at

Abstract. Composability and mixed criticality are concepts that promise an ease of development and certification for safety critical systems in all industrial domains. In this paper we define the necessary requirements, highlight issues and classify fault containment, when extending already existing triple modular redundant architectures with these concepts. We evaluate the needed adaptations and extensions of triplication mechanisms with respect to the required safety properties. Finally, we suggest novel architectures for serving triplicated modular redundant applications and compare them to the previously presented solutions.

1 Introduction

Failure of a safety critical system can result in harm to humans and the environment. To ensure that the resulting threat is acceptably low, the system has to be certified according to the applicable industrial standards. These standards define processes to classify systems in levels of criticality with respect to the potential damage they could cause when failing. SIL, ASIL and DAL are examples of classification schemes in standards of the railway, automotive and avionics domain. These levels define different processes and methods to be followed during a system's lifetime to keep its probability of failure acceptably low. These methods are then applied to the whole system and the whole system is certified. Should parts of the system change, substantial effort is necessary for re-certification, as the corresponding certification process has to be repeated for the whole system to demonstrate that safety is still guaranteed.

Composability aims to overcome this limitation of certification. Its key principle is to decompose the system into failure containment regions that provide sub-services independent from each other, even if – and specifically in case –

** Part of this research was funded by the ARTEMIS Joint Undertaking (nSafeCer, Grant Agreement number 295373), and the Austrian partners' national funding agency Austrian Research Promotion Agency (FFG) on behalf of the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT).

one of these should fail. In conjunction with an appropriate reasoning that the overall system service is represented by the composition of these sub-services, it becomes possible to (somewhat) move the certification focus to the failure containment regions. Of course, provisions have to be taken to establish this failure containment. On this foundation different criticality levels can be assigned to different sub-services (”mixed criticality system”), and upon changing a sub-service, re-certification can be limited to the latter, rather than having to stick to a monolithic system-level view. Naturally this is most beneficial when using easily changeable sub-services in physical dependency, like software-implemented sub-services executing on the same hardware. Later on we will show that composability can also improve the hardware utilization for the whole system.

Triple modular redundancy (TMR) is a wide spread approach in the industry to build fault-tolerant systems using three fault containment regions[3]. Depending on the specific TMR architecture, the applicable fault hypothesis can range from random transient hardware faults to systematic design faults. TMR covers techniques from triplicating gates within an integrated circuit, to triplication of sensors and displays, where the human decides in the process. In this paper we will concentrate on TMR methods for triplicating software and investigate how the concepts of composability and TMR can be beneficially combined.

After a brief survey of related work in the next section, Section 3 will be concerned with the concepts and requirements of composability, and Section 4 will add the fault tolerance aspect to the discussion. On this foundation we will systematically review contemporary TMR approaches in Section 5. Finally we present new types of TMR architectures that take full advantage of the composability concepts in Section 6 before concluding the paper with Section 7.

2 Related Work

Concepts for mixed criticality and composability are already used in the industry. The avionics domain has adopted the concept of integrated modular avionics (IMA) for the integration of different safety critical components on one hardware/software platform. Its foundation is the ARINC report 651-1 [1]. The ARINC 653 standards define an application software standard interface for integrating software functions of mixed criticality on a common platform [2]. These standards are supported by industrial products for IMA, e.g. VxWorks 653. AUTOSAR is an approach to define a platform standard for the automotive domain. This includes a common interface for electronic control units and for allowing software reuse by providing a runtime environment for applications [3]. With the ISO 26262 standard the concept of SEooC (Safety Element out of Context) can be applied to certify a safety element in isolation, using assumptions of the operational context. The final evaluation is performed when the safety element is used in a specific system, and it includes verifying the correlation of the assumed context to the specific context within the system [4]. For the railway domain the

[3] Notice the difference between fault containment and failure containment; detailed definitions of these terms will be given later on.

CENELEC EN standards [5] provide generic safety cases for incremental certification, which should be suitable to construct a safety case for composability.

In [6] a time-triggered System-on-Chip architecture is presented that aims to achieve composability by hardware means. Another hardware implemented solution for composability, using different scheduling strategies for each resource, is presented in [7], and the technique of virtualization has been applied to it in [8].

Apart from industrial standards, the concept of software partitioning is discussed in [9] with the introduction of a separation kernel that further evolved to the MILS separation kernel [10]. Separation kernels of this type are usually based on microkernels, which also use partitioning [11]. Another prominent separation approach is the use of hypervisors, also called virtual machine monitors [12]. As discussed in [13], the precise border between microkernels and hypervisors is not that clear. A comprehensive state of the art in embedded virtualization can be found in [14]. Using virtualization for implementing a primary-backup fault tolerant system has been suggested in [15]. Different methods for virtualization and the concept of hardware virtualization support are presented in [16].

The general concept for software-implemented fault tolerance was introduced by Wensley in [17]. An overview of methods for achieving fault tolerance with replication is given in [18], covering triple modular redundant architectures with hardware lock-step, as well as software-only solutions on COTS hardware.

3 The Concept of Software Composability

Safety is a system property, therefore a single system component can only fulfill a safety property within the context of the whole system application [19]. The intention of composability is to allow building safe and certified systems by careful integration of components, some of which provide safe and (pre)certified functions. As an immediate advantage this facilitates the reuse of certified components. We call such a component *function-set* (FS), to emphasize that functions are provided by one or more entities, especially in a TMR architecture (see later). These FSs are then deployed within an *integration environment* (IE) to build the whole system. Here the possibility of sharing the same IE for different systems, thus saving cost, space and energy, represents another advantage.

A FS provides a (sub-)service within the application context and is assigned a criticality level according to the criticality of that service. Clearly, the proper provision of this service can only be guaranteed on the condition that the IE exhibits all properties that have been assumed in the design of the FS. While this is relatively trivial to establish in the traditional federated architectures (i.e. using a separate IE per FS), it becomes an issue in integrated approaches, since the properties of the IE, as perceived by a single FS, are (dynamically) influenced by the other FSs during their execution. Therefore, to enable composability, every FS must be associated with an appropriate *function-set contract*, specifying its requirements to the IE for correct execution. We refer to the deterministic

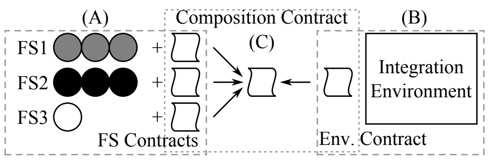

Fig. 1. Mixed criticality and composability certification strategy.

availability of resources from the IE as *predictability*[4]. This first constituent of composability becomes crucial when FSs or elements of the IE are to be changed. Notice that in the interest of a simple FS contract static guarantees (i.e. high predictability) are beneficial, while more fine-grained, even dynamic, requirements usually facilitate a better resource utilization. In addition, the former is easier to enforce by technical means (see later).

The second constituent of composability, namely *non-interference* concerns undesired effects that the execution of a FS may have on the IE and consequently on other FSs, specifically in case of failure. Again one could, in principle, conduct a fine-grained, application specific analysis on malign and non-malign cases to allow for the largest freedom. In practice, however, the most rigorous approach has proven most effective – a strict *failure containment*[5]. Herein, each FS forms an individual failure containment region, whose failure remains local and has no effect on any of the others. This task has to be fulfilled by the IE which needs to provide technical provisions to *separate* the FSs from each other.

Ultimately, the composition approach allows to split the certification of a system into three parts, as illustrated in Figure 1:

(A) The safety critical FS is certified with respect to its FS contract, which specifies all the FS's requirements.
(B) An IE, e.g. hardware boards and middleware, is certified with its provided properties and requirements, stated in an integration environment contract.
(C) The FS- and IE contracts are specified using generic properties, like network bandwidth, to enable reusing of FSs in different IEs. A concrete system is then certified by matching the IE contract with the FS contracts in a *composition contract.*

Please note that for each safety critical FS step (A) is performed separately, as well as step (B) for each specific integration environment. Furthermore, for each new or altered system step (C) is done. This method needs more initial effort than certifying one system as a whole, still it is more efficient when building several slightly different systems, or altering existing ones. Additionally, a good utilization of the hardware resources within the IE is expected.

The use of a common IE introduces unwanted dependencies between FSs. This is why composability requires specific attention and, ultimately, specific provisions for partitioning. The general idea is to implement a composability layer

[4] Unlike [20] we define predictability with respect to available resources for FSs and not as predictability of execution times and resource demand.
[5] Like the "Gold Standard for Partitioning" in [21].

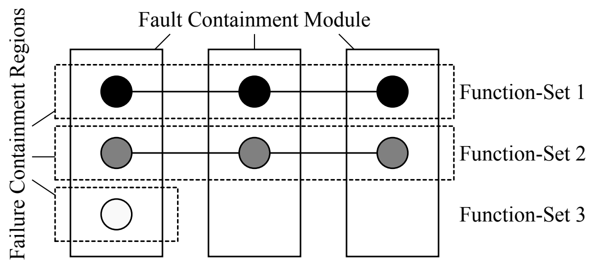

Fig. 2. Function-sets in failure containment regions and fault containment modules.

that provides failure containment regions (*partitions*) within the IE, independent of the specific hardware setup. A partitioning concept for fail-operational systems is presented in [21]. Here the correct and timely execution of all safety-critical FSs is mandatory, and the system must remain operational even under (the hypothesized) faults.

Safety critical systems with a safe state, in contrast, can handle the case where no results or outputs are provided by a safety-critical FS. The important property here is that no *incorrect* outputs are produced. This is normally ensured by fault-tolerance measures like a TMR architecture, and the remaining, but very important, requirement on the partitioning layer is not to undermine the error detection and/or masking capabilities of these measures, e.g., by introducing common-mode failures. Beyond that, the failure containment regions do not require as strong separation as in the fail-operational case, especially wrt. scheduling and timing. For example, it may be tolerable to guarantee resource access with some probability. In the remainder of this paper we will use the term *composability layer* rather than partitioning layer to emphasize that it is not necessarily required to achieve full partitioning in all cases.

4 Combining Composability and TMR

The primary goal of TMR is to keep the system operational in case of a single random hardware failure. The principle is to mask the output of one failed module by the outputs of the remaining two modules. Consequently, the architecture is separated into three fault containment modules, and it is essential that only one fails at a time. There are three threats to this principle: (1) In case of *near-coincident faults* two (or all) modules fail due to faults of independent origin, which in theory is ruled out by the single-fault assumption, and in practice, the very low fault rates make this extremely improbable. (2) In case of *common cause failures*, we again encounter failures of two (or all) modules, this time, however, these originate in the same single fault. That is why fault containment between the modules is so important. (3) In case of *spare exhaustion*, one replica did not recover from a previous fault and therefore, there are too few modules available to mask the current fault with the remaining replicas. This makes *recovery* of a failed module essential.

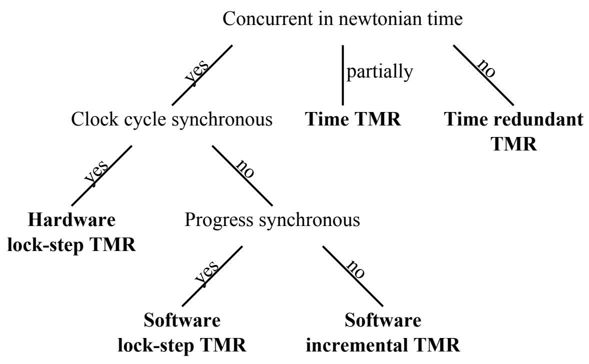

Fig. 3. TMR Classification for Software Triplication.

Composability is an orthogonal concept that aims, as already outlined in Section 3, at achieving better resource utilization and ease of the certification process upon integration. These benefits equally apply for TMR architectures. As illustrated in Figure 2 the IE may comprise replicated modules, and we have two orthogonal containment regions in a composable TMR architecture:

- The replicated hardware modules form fault containment regions required to prevent common cause failure of the TMR architecture. With a properly working TMR, the safe execution of a FS can be ensured even in case of a random fault in its IE.
- Within these modules, each FS forms a failure containment region. This establishes the non-interference required for composability. With non-interference, the safe TMR execution of a FS in presence of other FSs is guaranteed.

In this scheme a FS comprises three entities, each representing a computing channel. FS 1 and FS 2 are examples for this. Assume the fault containment modules are independent hardware boards, then the failure of one is observable as fault of one entity for FS 1 and 2. In contrast, if FS 2 fails due to a software error, the failure containment regions provide protection for FS 1 and FS 3.

Note that in our example FS 3 comprises one computing channel only, as it is not safety-critical. This already indicates that having three computing channels per FS only illustrates the fundamental principle of this architecture, and many variations are possible. For the fault tolerance scheme, e.g., simplex or duplex architecture could be chosen instead of TMR as well, as is appropriate for the needs of the specific FS. More generally, there is a lot of freedom in aligning the failure containment regions of the FSs with the modules' fault containment regions. Exploring this solution space will be the topic of the next sections.

5 Contemporary TMR Architectures

TMR methods for replicating software can be classified as shown in Figure 3. They differ in properties of fault containment, concurrency, synchrony and resource utilization. In the following we discuss these properties, as well as recovery and how the composability concept can be introduced in these currently available TMR architectures.

5.1 Time redundant TMR

For Time redundant TMR, software instructions are triplicated at compile time and voting instructions are added automatically. The triplicated instructions use different memory, which is also assigned during compilation. The fault containment "modules" in this architecture are instruction sequences together with their memory. This method does not require special hardware and can be used in a COTS processor. As Time redundant TMR uses only one processor, it can only mask transient hardware faults, e.g. SEUs. In this architecture "recovery" is performed by simply masking the erroneous output value and using the voted one as input for the next instruction triple. This creates a significant overhead for voting. Naturally, repairing and replacement cannot be performed during the operational phase of the system.

Note that here the potential conflict between fault tolerance and composability becomes apparent: Separation of memory and CPU can be ensured by a composable scheduler and a MMU, respectively. In this setting, however, both, the scheduler, as well as the MMU represent single points of failure from the fault-tolerance point of view. While the scheduler (as well as potential further software-based composability services) can, just like the FSs, be protected by time redundant execution as well, the MMU remains problematic.

In general, the performance impact can be deducted from the scheduling scheme. With a static cyclic scheduler, the reaction time can be derived from the maximum time between scheduled slices of the safety critical FS entities and the slice width. However, this can vary for specific FSs and also depends on the shared I/O devices. Mixed criticality can achieve good hardware utilization in this architecture, as only safety critical FSs are triplicated (at compile time), while non-critical FSs can use CPU and memory at native speed.

The patented *Time TMR* [22] architecture has the same properties as Time redundant TMR. The only difference is that some of the instructions are executed on different CPU components and by this it might be possible to mask a permanent hardware failure in a specific part of the CPU. This also reduces the overall execution time in comparison to Time redundant TMR.

5.2 Hardware lock-step TMR

Hardware lock-step architectures are widely used in the industry [23]. This "classical" architecture tolerates both transient and permanent hardware faults. It comprises triplicated CPUs and memory with hardware voters in between. The CPUs operate with the same clock (which is why the CPUs are often located together on one hardware board) and memory operations are voted and corrected on error, thus voting is performed purely by the hardware. As for recovery, the state of one CPU can be recovered from the other two by halting the execution of the triple and reconstructing the state of the erroneous CPU from the other two. A current industrial example is the D602 board from MEN Mikro Elektronik.

The inherent triplication of the lock-step architecture is advantage and drawback in the composability context at the same time. Safety critical FSs, as well as non-critical FSs are all automatically triplicated and recovered, which degrades resource utilization. Fulfilling the composability requirements from Section 3 for each fault containment unit locally is already sufficient for composability of the triplicated modular redundant system, since the triplication mechanism is not influenced by the composability layer – rather the composability layer is triplicated. Like for Time redundant TMR, the performance impact is closely related to the composable scheduling and I/O sharing strategy.

5.3 Software-based TMR

The two software-based TMR methods discussed here, Software incremental TMR and Software lock-step TMR, share most of their properties. Both tolerate transient and permanent hardware faults, and possibly some quasi-random software faults as described in [24]. In both architectures a middleware software is running on different hardware boards, which provides voting and synchronization to applications built on top. A FS in this architecture consists of all three entities of the application and TMR middleware. These applications must follow design constraints provided by the middleware to guarantee replica-deterministic execution. Contrary to Hardware lock-step TMR it is possible to use COTS hardware boards in this architecture, and maintenance actions can be performed online during system operation. The difference is that during the synchronization phase in Software lock-step TMR, replica-deterministic applications cannot make any progress and must have processed all their input data and generated the respective output before synchronization; whereas Software incremental TMR has no such constraints and therefore less stringent requirements on execution times.

Synchronization can be performed event-based or periodically. Periodic synchronization has a minimum and maximum period in which any of the modules can initiate the synchronization phase. The event-based synchronization is triggered when I/O is available and has to be voted. Even in the event-based case a minimum period (to avoid overload) and a maximum period (to check for liveness) are required, which finally comes down to periodic synchronization. Periodic synchronization can, in turn, be interpreted as event-based synchronization with periodic events. In any case, synchronization is achieved by exchanging messages between the middleware instances. Thus, message transmission time and synchronization event latency, the time it takes from the occurrence of an event to its processing in the middleware, are the key performance values.

In both architectures recovery has to be performed by the middleware without interruption of the active replica. For this purpose the middleware has to know which replicated data needs to be recovered and has to have enough communication resources in addition to the regular synchronization requirements.

Figure 4 shows the introduction of the composability layer to software-based TMR below the triplication middleware, the TMR layer. While this enforces the

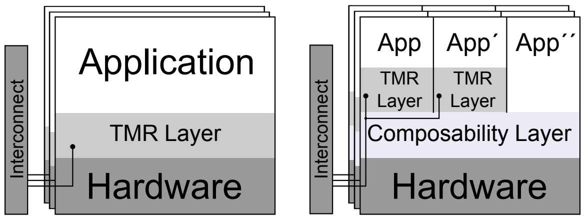

Fig. 4. Software-based TMR without (left) and with (right) composability layer.

desired separation of the TMR instances, the execution of the latter now relies directly on the composability layer, more specifically its scheduling and the provided communication resources. This has a significant effect on the TMR synchronization process, resulting in an increased maximum synchronization event latency as well as message transmission time and jitter of both. Low latency and short message transmission time may be achieved with small scheduling periods for safety critical FSs, which increases the overhead for scheduling as more context switches are needed. Message transmission times also get longer due to other TMR middleware using the same communication paths and interfaces in parallel. Bandwidth guarantees can limit the influence on these paths, however message transmission times are increased in any case. These effects cause the duration of the synchronization phase to increase and also have a larger jitter, which in turn means that applications can have shorter computation time within the same period. Thus, more computational reserve has to be given for a Software lock-step TMR to guarantee completion in between synchronization phases. This is not as strict for Software incremental TMR, but the timeouts on application level may need to be reconsidered. Also the communication resources for recovery have to be available, which is not as latency sensitive as synchronization.

Fail-safe systems might also be able to tolerate configurations with less computational reserve, where resource shortage can only result in loss of availability and not safety. However, composability and mixed-criticality can achieve good resource utilization, as non-critical FSs are not replicated. Furthermore, if an error is detected, it might be sufficient to only reboot the affected partition and not the whole hardware board.

6 TMR Architectures Leveraging Composability

6.1 Static TMR-Composable Architecture

In the software-based TMR architecture described above, non-triplicated FSs can be added, as long as there are enough free resources on one of the hardware boards. For TMR FSs all three boards need free resources. If only one of them has insufficient computational resources, a complete new hardware triple must be added. With composability it is now possible to introduce a new kind of architecture, the static TMR-composable architecture. Replica and their communication can be statically assigned to any hardware board and communication links inbetween and are no longer fixed by a hardware triple. The composability and triplication mechanism are the same as for software-based TMR and also COTS hardware boards and network equipment can be used. This method improves resource utilization and scalability of the whole system. Figure 5 illustrates a possible scenario, where adding one TMR entity to a system can be achieved by adding only one hardware board, provided that the communication links have enough bandwidth. The graphic does not explicitly show the required additional interconnect. This can either be solved with several direct links, or a redundant network, e.g. two switches each connected to one of two Ethernet interfaces of a board. This change of the connections properties has to be accounted for in the composability model. Furthermore, the impact of the new networking components on the availability must be acceptably low in a redundant configuration. Maintenance can still be performed without system downtime.

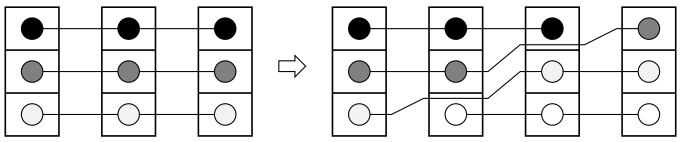

Fig. 5. Change of static deployment when adding a new triple and hardware node.

6.2 Dynamic TMR-Composable Architecture

In contrast to all previously presented TMR architectures, the Dynamic TMR-Composable Architecture has no static mapping of replica to hardware boards. A FS manager is dynamically deploying safety critical and non-safety critical FSs in a cluster of redundantly connected hardware boards. This cluster can be built from COTS hardware boards and network equipment. The FS manager starts and stops partitions, establishes virtual links inbetween them and can even implement load balancing. Starting, stopping and interconnecting safety critical FS entities is a safety critical task, thus the FS manager has to be implemented as safety critical component within the Dynamic TMR-Composable Architecture itself. This management of safety critical components is actually subject to certification. It has to be performed with building a valid composition contract from the component and environment contracts. To automate this process, the contracts have to be formalized in a machine readable way. For fail-safe systems, it may be sufficient to implement mechanisms for the individual triples to check whether their component contracts are fulfilled, otherwise they can trigger a safety reaction.

The composability layer is the same as for software-based TMR. However, as the composability policies concerning bandwidth guarantees and scheduling are dynamically changed by the FS manager, it has to be ensured that such changes are covered in the composition contract. Resource usage, maintainability and availability are improved in comparison to the Static TMR-Composable Architecture, since replicas can be moved between hardware boards and a suitable trade-off for the whole system could be found for all those factors.

6.3 Software-based TMR using Separation

Composability provides failure containment regions. These can be exploited as fault containment modules for a software-based TMR architecture[6], resulting in a fault tolerance similar to Time redundant TMR with overhead and software restrictions of software-based TMR. However, it is possible to use COTS hardware and reuse existing triplication middleware and possibly already implemented for this middleware. Additionally, fault containment could be improved with the use of multi-core CPUs.

7 Conclusion

We have taken a closer look at the idea of combining composability with TMR. Our analysis has shown that these concepts are orthogonal in that composability requires failure containment regions, while TMR is based on fault containment module. There are multiple ways of combining these properties. We have aligned the existing TMR architectures within this solution space and systematically identified their benefits and needs when augmented with a composability layer. On top of that, we have proposed novel TMR schemes that specifically leverage the existence of composability for making more efficient use of the resources.

For software-based TMR, as well as our newly proposed architectures we have identified the criticality of scheduling and communication for performance. We do have some measurement results available from a practical industrial system (not presented in this paper) that allow to quantify this dependence. As part of future work we plan to extend and generalize these measurements.

References

1. Committee, A.E.E., et al.: Arinc report 651-1: Design guidance for integrated modular avionics. Aeronautical radio, Inc., Annapolis, Maryland (1997)
2. Prisaznuk, P.: Arinc 653 role in integrated modular avionics (ima). In: IEEE/AIAA 27th Digital Avionics Systems Conference (DASC 2008)., IEEE (2008) 1–E
3. Bunzel, S.: Autosar–the standardized software architecture. Informatik-Spektrum **34**(1) (2011) 79–83
4. Espinoza, H., Ruiz, A., Sabetzadeh, M., Panaroni, P., et al.: Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems. In: Software Certification (WoSoCER), 2011 First International Workshop on, IEEE (2011) 1–6
5. CENELEC, E.N.: 50126-railway applications: The specification and demonstration of reliability, availability, maintainability and safety (rams). European Committee for Electrotechnical Standardization (1999)

[6] assuming the failure containment mechanisms provide sufficient fault containment

6. Kopetz, H., El Salloum, C., Huber, B., Obermaisser, R., Paukovits, C.: Composability in the time-triggered system-on-chip architecture. In: SOC Conference, 2008 IEEE International, IEEE (2008) 87–90
7. Hansson, A., Goossens, K., Bekooij, M., Huisken, J.: Compsoc: A template for composable and predictable multi-processor system on chips. ACM Trans. Des. Autom. Electron. Syst. **14**(1) (January 2009) 2:1–2:24
8. Molnos, A., Milutinovic, A., She, D., Goossens, K.: Composable processor virtualization for embedded systems. In: Proceedings of the Workshop on Computer Architecture and Operating System Co-Design (CAOS)., Springer (2010)
9. Rushby, J.: Design and verification of secure systems. In: ACM SIGOPS Operating Systems Review. Volume 15., ACM (1981) 12–21
10. Alves-Foss, J., Oman, P., Taylor, C., Harrison, W.: The mils architecture for high-assurance embedded systems. International Journal of Embedded Systems **2**(3) (2006) 239–247
11. Tiwari, M., Oberg, J., Li, X., Valamehr, J., Levin, T., Hardekopf, B., Kastner, R., Chong, F., Sherwood, T.: Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In: ACM SIGARCH Computer Architecture News. Volume 39., ACM (2011) 189–200
12. Masmano, M., Ripoll, I., Crespo, A., Metge, J.: Xtratum: a hypervisor for safety critical embedded systems. In: Proceedings of the 11th Real-Time Linux Workshop. Dresden. Germany. (2009)
13. Heiser, G., Leslie, B.: The okl4 microvisor: Convergence point of microkernels and hypervisors. In: Proceedings of the first ACM asia-pacific workshop on Workshop on systems, ACM (2010) 19–24
14. Gu, Z., Zhao, Q.: A state-of-the-art survey on real-time issues in embedded systems virtualization. Journal of SW Engineering and Applications **5**(4) (2012) 277–290
15. Bressoud, T., Schneider, F.: Hypervisor-based fault tolerance. ACM Transactions on Computer Systems (TOCS) **14**(1) (1996) 80–107
16. Adams, K., Agesen, O.: A comparison of software and hardware techniques for x86 virtualization. In: ACM SIGOPS Operating Systems Review. Volume 40., ACM (2006) 2–13
17. Wensley, J.: Sift: software implemented fault tolerance. In: Proceedings of the December 5-7, 1972, fall joint computer conference, part I, ACM (1972) 243–253
18. Poledna, S.: Replica determinism in distributed real-time systems: A brief survey. Real-Time Systems **6**(3) (1994) 289–316
19. Leveson, N.: Safety as a system property. Communications of the ACM **38**(11) (1995) 146–
20. Akesson, B., Molnos, A., Hansson, A., Angelo, J., Goossens, K.: Composability and predictability for independent application development, verification, and execution. In: Multiprocessor System-on-Chip. Springer New York (2011) 25–56
21. Rushby, J.: Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document (2000)
22. Czajkowski, D., McCartha, M.: Ultra low-power space computer leveraging embedded seu mitigation. In: Proc. IEEE Aerospace Conf. Volume 5. (2003) 2315–2328
23. Witwer, B.: Systems integration of the 777 airplane information management system (aims): a honeywell perspective. In: Digital Avionics Systems Conference, 1995., 14th DASC, IEEE (1995) 389–393
24. Gerstinger, A.: Runtime diversity against quasirandom faults. In: Systems, 2009. ICONS '09. Fourth International Conference on. (March) 145–148