Embedded multi-core systems for mixed criticality applications in dynamic and changeable real-time environments

Project Acronym:

EMC²

Grant agreement no: 621429

Deliverable no. and title	**D4.11 Verification methods, techniques and processes for mixed-criticality applications on multi-core systems**	
Work package	WP4	Multi-core Hardware Architectures and Concepts
Task / Use Case	T4.5	Processor qualification approach and verification
Lead contractor	TVS	
Deliverable responsible	University of Bristol, Bristol - UK Samuel Pagliarini, sam.pagliarini@bristol.ac.uk	
Version number	V1.0	
Date	02/04/2015	
Status	Final	
Dissemination level	PU	

Authors

Participant no.	Part. short name	Author name	Chapter(s)
94	UoBR	Samuel Pagliarini	All
94	UoBR	Simon Hollis	All
94	UoBR	Kerstin Eder	All
97	TVS	Michael Bartley	All

Document History

Version	Date	Author name	Reason
v0.1	19/01/2015	Samuel Pagliarini	First draft
V0.2	23/01/2015	Samuel Pagliarini	Sections 1, 2, and 3 done
V0.3	27/01/2015	Samuel Pagliarini	Sections 1-5 done
V0.4	06/02/2015	Samuel Pagliarini	Several paragraphs rephrased to reflect Michael's/Kerstin's comments
V0.5	22/02/2015	Samuel Pagliarini	Added content on complexity & breakdown of methods per level
V0.6	09/03/2015	Samuel Pagliarini	Added a new section (4.7) and minor changes in other sections
V0.7	17/03/2015	Samuel Pagliarini	Draft for internal review
V1.0	27/03/2015	Samuel Pagliarini	Finalised version
	02/04/2015	Alfred Hoess	Final editing and formatting, deliverable submission

Table of contents

List of figures

1 Introduction

Figure 1.1 shows how the design intent, the design specification, and the design RTL code are related to each other to compose the space of design behaviours [1]. Two overlapping processes, namely validation and verification, can be seen in Fig. 1.1. Validation can be understood as the process of matching intent with the specification and implementation. Validation is concerned with assurance and compliance, a process that requires understanding abstract customer's needs.

Verification, on the other hand, is a more concrete task. The primary goal of verification is to establish confidence that the intent declared in the specification was preserved by the implementation [2] [3]. This is a broad definition that captures the essence of verification: try to prove that what was designed is indeed what was planned and intended.

Another way to look at verification is as an approach to maximize region *G* from the image below. This is accomplished by carefully combining methods, techniques and methodologies, a process that ultimately tries to minimize all the other undesired regions (*A-B-C-D-E-F*).

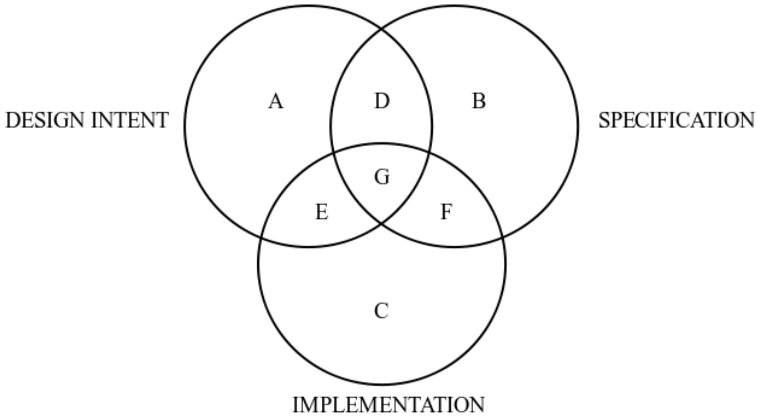


Fig. 1: Design intent, specification and implementation: the space of design behaviours [1]

Maximizing *G* is by no means an easy task. A fundamental issue faced by verification is that the design space is comprised of distinct elements. For instance, the design is usually described in some form of Hardware Description Language (HDL), while the specification can be a simple written document. The process of 'translating' a written document to a hardware description is especially susceptible to misinterpretations. The intent itself can be even harder to formalize, thus 'comparing' these three elements is very challenging.

The design, also commonly referred as Design Under Test (DUT), is the object being verified. The complexity and size of such DUT can be as simple as of one stand-alone RTL module, as well as a system with hundreds of such modules, interconnected in a variety of ways, as a typical System on Chip (SoC) is. Verifying such a system can also be achieved by breaking down into different levels with different verification goals (i.e., unit level versus system level verification). However, the premise is still the same, i.e., achieve confidence that the design matches its specification.

The sheer complexity of an SoC creates a difficult circuit design problem, and an even more challenging verification problem. This is referred as the v*erification gap*, a difference between the ability to design and to verify circuits.

Ideally one would like to be able to list all possible functionalities of a design and then make sure that they are all implemented. This is not enough from a verification completeness point of view, as these functionalities have also to be proved to work together in all possible configurations of interest. The number of such configurations can be extremely high, if not intractable. Another issue is to measure if these configurations were actually exercised, which is no simple task either. Verification progress is usually measured by indirect means such as coverage. This difficulty to measure progress adds another whole layer of difficulty to verification.

Complexity is a recurring issue with verification and it will be highlighted multiple times in the text that follows. Although the complexity of a system can be given by rough measurements like number of gates/modules, this figure does not account for how these entities interact.

On top of the aforementioned issues, there are also new challenges that come from new systems and technologies. For instance, (partial) reconfigurability is a reality in modern Field Programmable Field Arrays (FPGAs), which clearly creates a challenge: how to verify a system that is not static? How to verify that the system as a whole remained working while some portions were undergoing reconfiguration?

Other concerns such as low power and/or performance are also of interest: how to verify that a system with power management features operates properly while keeping performance at an acceptable level? These concerns extend the notion of what a design's intent is and what the design specification should contain.

An increasingly important trend in the design of real-time and embedded systems is the integration of components with different levels of criticality onto a common hardware platform. Criticality is a designation of the level of assurance against failure needed for a system component. A Mixed Criticality System (MCS) is one that has two or more distinct levels (for example safety critical, mission critical and non-critical) [4]. At the same time, these hardware platforms that support MCSs are migrating from single cores to multi-and many-cores. This type of many-core MCS is the focal interest of the EMC2 project and this document covers the verification aspects of such systems.

All these trends combined – but especially many-core based MCS – create a scenario in which the state of the art in verification might not be advanced enough. The huge complexities that arise from interaction of processors, the interconnects themselves and between memory accesses, all in the context of mixed criticality applications, potentially require the development of new verification methodologies that extend the state of the art beyond the current SoC level. Development includes both static and simulation-based approaches and combinations of these to arrive at a methodology that is effective and efficient in practice, and paves the way to certification.

1.1 Objective of this document

This document provides the context for verification work in the EMC2 project. It addresses the verification needs of partners by:

1. Surveying existing state-of-the-art methods, techniques and methodologies for verifying systems.
2. Listing the potential verification challenges applicable to EMC2-specific systems and their requirements.

3. Identifying key partner verification needs and their current techniques for solving these needs.
4. Highlighting verification areas that are insufficiently covered according to the needs of the project and relevant partners.
5. Sign-posting topics for further investigation during months 13-24 of the project.

1.2 Related deliverables

In M24, a second verification related deliverable (D4.12) will be created, which will address tools (and their associated methodologies) that tackle the issues and challenges sign-posted in this document. Therefore this document does not cover the tools being currently used by the industry.

Inputs to this document are D4.1 (Mixed Criticalities Requirements Report) and an internal survey of architectures being currently used. The outcome of this survey is annexed.

2. Verification Methods

In general, verification of a design has been accomplished following two principal methods, simply referred as static and dynamic verification [1]. Regarding the dynamic approach, one usually is referring to a set of simulation-based techniques that may include all of the following:

- random, constrained-random, or directed stimulus [5]
- assertions [6]
- code coverage [7]
- functional coverage [8]
- transactions & sequences [9]

Regarding static verification, one usually is referring to a set of methods that includes one of the following:

- theorem proving [10]
- equivalence checking [11]
- property checking / formal assertion checking [12]

Dynamic verification is mainly simulation-based and it is by far the preferred method being used by the industry. The reason for the large adoption of a simulation-based approach is that, although new methodologies that benefit from formal and semi-formal methods have been proposed and adopted by the industry, these formal methods are still scope-limited [12] [13]. Formal methods, however, have found widespread use for specific verification applications. Figure 2.1 lists the different state-of-the-art verification methods and techniques which are detailed in the following sections.

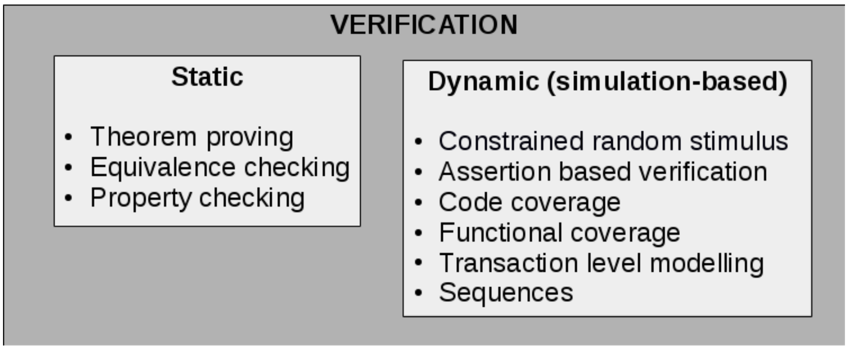


Fig. 2: Different techniques and methods used in verification

There are many challenges associated with the design and verification of traditional systems. However, the complexity of these systems seems to be dictated by the design while verification struggles to keep at the same pace [14]. This leads to an inevitable shift of priorities between verification and design, one that might be manageable or acceptable when dealing with consumer products. Nevertheless, the same might not be true for high dependability applications such as medical, space, and automotive. One approach to alleviate this problem is the introduction and adoption of DFV (Design for Verification) methodologies. The purpose of DFV can roughly be understood as to leverage a designer's intent and knowledge to strengthen the verification effort [2]. DFV can be achieved by defining tasks designers can easily perform that will increase verification quality or ease the verification process.

Assertion Based Verification (ABV) is an important aspect of DFV [1] [2] [3] [6]. Assertions are statements that capture design assumptions. These statements can be embedded into the design. ABV brings several benefits at different levels. For instance, capturing design intent (in a concise format) enables this information to be used to drive and automate formal verification, enabling the proof of certain design properties. Assertions at block level act as continuous checks during simulation, helping to assure that the block is free of bugs. Assertions are also used at the system level, especially to capture constraints on interfaces, thus facilitating a bug-free integration.

2.1 Static Verification

Static verification is a collection of techniques that do not rely on simulation. Formal verification is the static counterpart of simulation. Instead of applying carefully generated stimuli to a design like simulation does, formal verification tries to prove that the DUT operates correctly under all possible stimuli, therefore proving or disproving the correctness of a design.

Static verification uses formal methods or mathematics for achieving that goal and therefore is complete by nature, given that the modelling was accurate and representative. For instance, if a property of a design is proved successfully, it will be valid for every input scenario and every possible state scenario. However, proving such statement can be a cumbersome task that might involve achieving partial proofs through an user-guided process.

Static verification goes beyond proving the correctness of a design through property proving. There are examples where static verification techniques were and still are being used by the industry with a high degree of success. These are loosely termed *formal apps*. The most notable scenario is equivalence checking, typically done over different views of the same DUT, typically gate-level versus RTL [11]. Proving clock domain crossing solutions are appropriate is also a problem that can be solved with static verification [15] [16]. Other uses include power-aware formal verification, linting, and connectivity checks.

2.2 Dynamic Verification

Dynamic verification is of a simpler nature than static verification. Essentially it consists of building meaningful input scenarios, simulate them, make sure they have touched a certain functionality of the design, and check that the output was as expected. Due to its simplicity, simulation-based dynamic verification is the *de facto* industry verification method.

However, simulation struggles with achieving completeness. There are 'clever' ways to mitigate the issue and a few of them are mentioned in Section 3. It all boils down to the quality of the simulation being performed, which is a hard characteristic to quantify.

Multiple testbenches can be built in order to achieve a high quality verification. The term testbench has been used to describe varied solutions, ranging from a very basic set of pin-level stimulus run standalone in a simulator, all the way to a major part of a sophisticated verification environment containing complex models and interfaces. These environments and their terminologies change according to what methodologies are being adopted, but in general they can be broken down into three parts: input generation, checking schemes, and monitoring assets.

Higher quality verification can be reached by increasing the meaningfulness of the inputs created by input generators, as well as by increasing the quality of the sequences of inputs being generated. This difficulty of building meaningful inputs is known as the generation problem.

There are approaches that try to automatically guide the generation process [17], but in general this task relies on the expertise of the verification engineer, his knowledge of the design, and his knowledge of the current coverage holes that remain to be verified.

Dynamic verification can make use of a golden model when one is available, to which the design can be compared against. Having models that behave like entire SoCs is borderline infeasible but models of smaller portions of the system can be used for selective checking. The standard approach when using a model is to submit the same inputs generated by a testbench to both the design and the model. When a mismatch is found, the testbench should be able to detect that automatically. This process is referred as self-checking, and it is a fundamental part of a verification environment. Simply put, a checker has to be able to say if a certain testbench was a pass or a fail. On top of that, when a mismatch is detected, a (good) checker ideally has to try to pinpoint the error source accurately. When this is possible, the scope of the debug process can be reduced enormously.

Another important part of a verification environment is the ability to monitor progress, typically by collecting coverage figures. There are several different coverage metrics and some of these are listed in the following section. Regardless of the metric, the goal is to achieve 100% coverage of the scenarios of interest. Exhaustively listing these scenarios is not always possible, therefore the quality of a good verification depends on how accurate the scenarios of interest capture the overall system behaviour and how well these scenarios were exercised, i.e., covered.
Simulation-based verification can often be 'replaced' by its emulation counterpart. The verification goals are virtually the same, however the number of input scenarios that can be considered is usually much higher, potentially leading to a better verification outcome. There are drawbacks from using emulation as well. Emulating a model or an assertion means that they have to be synthesizable. Another additional challenge is that, In general, emulation has a lower observability than simulation. Additional debug features can be used to increase observability and minimize the problem.

3. Verification Techniques and Processes

The techniques listed below are commonly used by a single verification method only, which is mentioned for each technique. More important than the techniques, is the way they are combined to achieve a successful verification. This topic is discussed in the Section 3.8 when verification processes (typically termed methodologies) are examined.

3.1 Theorem proving

This technique is used with static verification. Theorem provers such as HOL [10] may be used. The tool that is used determines the exact nature and syntax of the models and theories, but in general this requires a version of the DUT translated into formal logic as well as a 'specification', also written in formal logic. The theorem prover helps with the deductions and some degree of automated tasks, but the user is generally required to think of the high-level verification strategies.

3.2 Equivalence checking

This technique is mostly used with static verification. It's most common use is to verify that different views of the same circuit do match. These views could be, for instance, pre and post synthesis.

3.3 Transaction Level Modelling

This technique is used with dynamic verification. Traditionally built low-level testbenches will directly drive the ports of a DUT with different data every cycle. And it is up to the verification engineer to provide such data, which can be a laborious task. However, with Transaction Level Modelling (TLM) the abstraction level is raised and the testbenches start to deal with higher level operations. TLM can be very useful when dealing with protocols. One can think of this technique as encapsulating operations in a repeatable way, similarly to what functions represent in the software domain. The ability to work at a higher level of abstraction can lead to increased productivity.

TLM can also be seen as a way to facilitate information exchange, which is achieved by separating details of communication among modules from the implementation itself. This is achieved through the use of communication channels that hide low-level protocols. The drawback of using TLM as such is that it may not accurately capture timing aspects of the low-level communication mechanisms.

There's also a secondary benefit obtained by using TLM as it facilitates the exchange of data between software and hardware layers, providing an early platform for software development. TLM is also useful if the reference model being used for checking does not take inputs of the same abstraction level that the design does (and this is typically the case). With the aid of a transactor, TLM can serve both the design and the model with the same coherent input data (such as send and receives).

3.4 Sequences

This technique is used with dynamic verification. Sequences are a mechanism to create an ordered set of inputs, or an ordered set of transactions. Different verification languages / methodologies will define different ways to express sequences. The goal of a meaningful sequence of inputs is to exercise design features of interest that are hard to reach. Additionally, ways to cover and randomize sequences can be very helpful to a verification engineer.

3.5 Code coverage

This technique is used mostly with dynamic verification. As previously mentioned, the measurement of verification progress is usually given by coverage metrics, which can be divided into code coverage (sometimes called structural coverage) and functional coverage. Although most of the metrics are simple and easy to understand, the way they relate to the actual hardware functionalities is neither simple nor direct. Block, expression and toggle coverage are examples of code coverage metrics with such behaviour. Certain certification processes may require specific code coverage values, e.g., DO-254 [18] requires 100% statement coverage at minimum.

Block coverage is the simplest of the structural metrics. A block of code is defined as one or more statements that always execute together. The principle of this metric is to measure if each block of the RTL code was simulated at least once. Block coverage supersedes statement coverage (in which the execution of each statement is evaluated). Guaranteeing that each block was simulated eliminates the presence of simple errors due to unstimulated logic. Also, this type of metric is useful for detecting potential dead code in the design.

Expression coverage is a mechanism that analyses logical expressions in depth. Each expression is factorized and then monitored during simulation runs. The goal is to check that different terms of an expression have the capability of controlling the outcome of the expression being evaluated. Expression coverage complements block coverage because it can be used to identify why a given block of code has been stimulated.

Finally, toggle coverage measures if every bit of every signal has been properly stimulated. This measurement is done by observing if a given bit has toggled from zero to one and vice-versa.

3.6 Functional coverage

This technique is used with dynamic verification. Essentially, functional coverage tries to list, combine, and exercise functionalities of the design. As mentioned above, code coverage reflects how thorough the HDL code was exercised. Nevertheless, code coverage is not enough. The reason is simple: most functional coverage requirements cannot be mapped into code structures. Whenever it is necessary to correlate events or scenarios, code coverage fails while functional coverage does not.

Functional coverage is usually divided into data-oriented and control-oriented. When performing data-oriented collection, sampling mechanisms are used to collect values of registers and/or variables of interest. These values can be cross collected as well, in order to capture scenarios of interest. Input coverage models record characteristics of the stimulus stream applied to the DUT. Output coverage models record characteristics of the response of the DUT to applied stimulus. Internal coverage models record behavior internal to the DUT. A fairly simple example of a functional coverage point is a (meaningful) register in the design of a processor, such as the one that contains the code of the instruction being currently decoded. A verification engineer can define a set of valid values/ranges for that register and coverage is then used to tell if those values/ranges were actually simulated.

On the other hand, when performing control-oriented collection, mechanisms like assertions and sequences are being analyzed to detect specific events that are of interest. Covering assertions means to verify that they were fired at least once. The checking behavior of the assertion will then tell if the design property was respected or not.

3.7 Assertions

This technique is used with both static and dynamic verification. An assertion is a statement regarding some expected behaviour of the design. Assertions behave as checks and can be embedded into the design early, i.e., at the design time. They can also be inserted later by verification engineers. They are used to verify assumptions about how a design should operate. Assertions are very appropriate for verifying protocol scenarios at interfaces. They are also used to express definitive properties about a system or module.

Besides enabling DFV as previously discussed, assertions can be perceived as providers of internal test points in the design, therefore increasing the observability and ability to debug it. Instead of comparing the system as a whole to a complex model, smaller monitors are put in place, therefore making strategically placed checks on important parts of the design. Additionally, assertions can increase the verification productivity and minimize the integration effort.

3.8 Random and constrained random stimulus

This technique is used with dynamic verification. When the logic's complexity grows, the verification space increases rapidly, yet it is still required to cover that space properly. This brings randomization into the simulation testbenches. The problem with fully random testing is that, even for simple designs, it can be an unbounded problem. On top of that, generating fully random stimuli may not be ideal as many non-legal inputs can be generated, thus wasting simulation time. To solve this problem, the industry introduced and adopted the concept of constrained-random stimulus. This concept is now an integral part of all modern hardware verification languages [19].

Despite small differences from language to language, the process of creating good stimuli through constrained-random generation usually relies on the verification engineer. The engineer's role is to write a set of rules for controlling how stimuli should be generated. This task is usually accomplished by giving ranges and priorities to those rules. The set of rules is submitted to a constraint solver that produces the inputs that end up being used.

3.9 Testbench standardization

Arguably, most of the improvements in verification came from the development and adoption of standardized testbenches, together with the automation aspects enabled by those. These standards are referred as methodologies, and an example of such standard would be the Open Verification Methodology (OVM) [9]. These methodologies promote standardized ways to build verification environments that can easily be reused, modularised, expanded and integrated.

Verification methodologies form the backbone of a verification strategy. They also deliver a set of libraries that automate standard verification tasks, such as building transactions, sequences, and scoreboards. Since different methodologies do use different naming conventions for their composing blocks, this section will use the same conventions used in OVM [9]. The image below depicts a typical testbench environment built using OVM.

The top-level component of Fig. 3.1 is called the environment. Its main purpose is to interconnect all the other components. It may also be in charge of some high level monitoring and driving, as represent by the component labeled 'Bus OVC'.

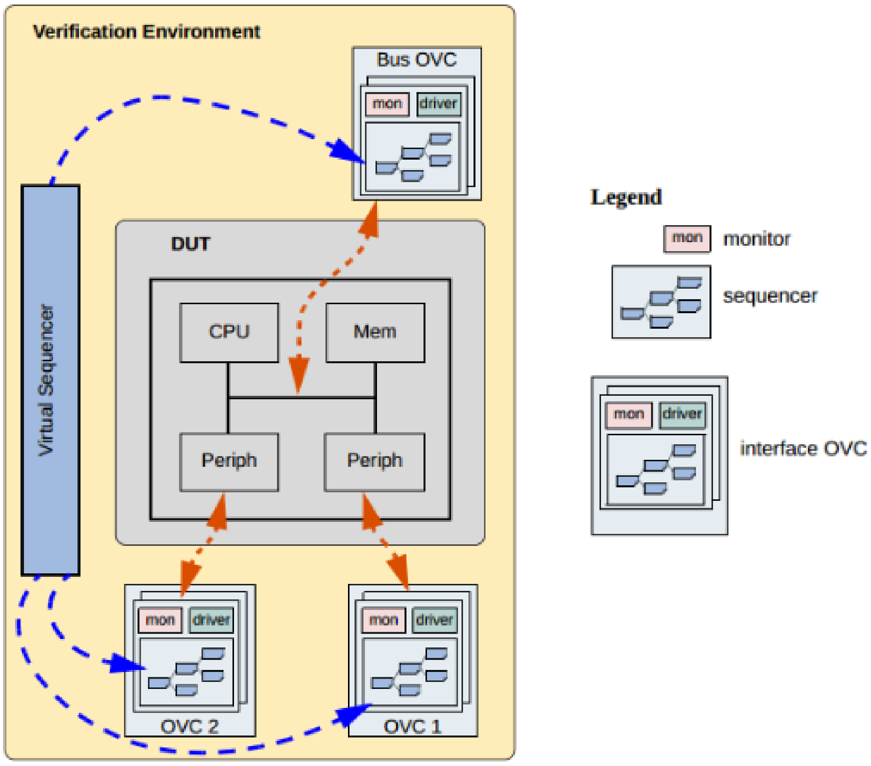


Fig. 3: Typical testbench environment built using OVM [9]

Typically, a verification environment will instantiate one or multiple agents. Examples of such agents are the components called 'OVC 1' and 'OVC 2' in Fig. 3.1. Agents are encapsulating components that group monitors, sequencers and drivers together. Agents can be active or passive. Active agents emulate devices and drive transactions according to test directives. Passive agents behave as monitors.

A monitor is a passive entity that samples DUT signals but does not drive them. Monitors collect coverage information from data items, extract events, and also perform checking. Even though reusable drivers and sequencers drive bus traffic, they are not used for coverage and checking. Monitors are typically used instead.

A driver is an active entity that emulates logic that drives the DUT, therefore drivers are responsible for knowing the DUT input/output protocol. A typical driver repeatedly receives a data item and drives it to the DUT by sampling and driving DUT signals. Data items are the lowest form of representing DUT input. Typically, constraints are applied on data items to shape the way they are created. Biasing the generation process allows to generate more legal input scenarios and also reach interesting corner cases.

Finally, a sequencer interacts with the driver to create ordered data items. Sequences capture the order between data items in user-defined sequences; these form a more structured and meaningful stimuli pattern.

There are issues with the scalability of a verification environment to cater for designs of varying complexities. It can be a very complex task to generate a full stream of meaningful inputs at the system level, or, even more, to have a system-level model to compare against. Therefore, a common practice is to split the verification effort into multiple smaller tasks that are easier to tract and cover the whole verification hierarchy.

Small modules can be handled in a standalone fashion first, typically by using a verification environment that is similar to the one in Fig. 3.1, however using a single driver and a single sequencer. Eventually, testbenches are built for larger blocks or subsystems, until system-level testbenches are built. Due to the complexity nature of what is being verified, testbenches at different levels can have different goals and use different constructions. For instance, higher-level testbenches have to stimulate and check integration between modules and subsystems, a concern that is not shared by low-level testbenches.

Coverage can also be collected differently depending on what the testbench at hand is simulating. For small modules, reaching 100% code coverage can be straightforward. The same might not be true for larger modules. Regression test suites can also be used to catch design changes that break the design. Another typical approach is to have sanity checks, which are usually small and directed testbenches that exercise simple scenarios and don't necessarily try to reach complex corner cases.

A lot of effort is also put into managing verification. Decisions have to be made during the verification cycle on where to focus effort. Assembling results from different testbenches and techniques is also important in order to have a clearer overview of the progress.

3.10 Verification management: bringing techniques together

Verification can be understood as a multi-faceted approach that combines several techniques and methods to reach its goal. Verification methodologies advocate good practices for combining techniques and reaching a successful verification. Verification projects typically utilise a range of techniques at different levels of SoC and software hierarchy, and generate a lot of data. Verification management therefore encompasses a wide range of tasks including:

- Defining a strategy: includes defining the verification techniques that will be deployed at the various levels of hierarchy, the sign-off criteria associated with those techniques and the data needed to determine success against those criteria.
- Planning: determining the resources needed to execute the verification strategy, the tasks, the timescales, etc.
- Managing execution: Collecting verification data, monitoring progress against plan and taking actions to keep to plan (or re-plan if required).
- Sign-off: Determining the readiness for sign-off. This should include comparing the actual verification data against the targets.

4. Verifying EMC2 systems: the challenges of complex mixed-criticality SoCs

The verification methods and techniques presented so far are of straightforward use if the target verification DUT is an isolated system. A key challenge of WP4.5 is to show how the same techniques may apply to the complex interactions that occur in MCS and systems with many-cores, especially when both are combined. The challenge is twofold as the techniques have to manage the increase of complexity that is inherent to an SoC, but they also have to cope with specific challenges that go beyond sheer complexity. These specific challenges are discussed in the text that follows.

4.1 Scenario: mixed criticality

Verifying MCS is, essentially, to deal with applications that have different levels of assurance against failures. Mixed criticality management in embedded systems is important, especially in domains such as automotive, medical, aviation, and space. Verifying MCS means dealing with this seamlessly, managing the use of resources by the hardware/software and also the partitioning of these systems as to guarantee no overlaps or undesired interactions between tasks and/or cores. Ultimately, verifying these types of interactions do not occur – or occur in a controlled way – is challenging.

A secondary but equally important issue is how verification feeds the certification mechanisms required for MCS. In the context of many cores and multi-threaded scenarios, MCS concerns such as coherence and non-interference arise.

4.2 Scenario: many cores

Due to increasing performance and (low) power demand, a shift in architectures has been seen, from multi-core to many-core architectures. Simply integrating multiple complex cores on a die is barely an option. Instead, integrating lots of smaller cores becomes an interesting option. Each small core delivers lower performance than a large complex core; however, the total compute throughput of the system can be much higher.

From a verification point of view, the shift to many cores can cause a significant increase of the verification effort due to the underlying complexity of a many-core system. Let us have a look at what this complexity looks like by breaking it down.

The verification of an isolated single threaded core can already be taxing since modern cores can have several advanced functionalities (check Section 4.5 and the appendix). This complexity can be further augmented if the cores support multithreading. It is not a matter of simply adding another functionality to the design; the interactions between threads have to be taken into account. Examples of arising concerns are the use of resource sharing, security extensions, coherence, non-interference and/or separation.

The adoption of many cores is, again, an extension of these concerns. One can think of the space state that needs to be explored when referring to a single core, versus the space state of a many-core system. The latter is clearly much larger, likely orders of magnitude larger.

In general, adding more cores to a system means creating more interaction, leading to a higher verification effort. However, these interactions can be scoped in such a way that intended interactions can be defined and verified. In practical terms, these cores have a 'binding contract' between them and a huge effort is spent to verify such contract is met. Verifying cache coherent systems is an example of verifying such defined behaviour.

Some of the interactions between cores are so complex to describe and model that they nearly become of a chaotic nature if no restrictions are applied. Verification of the behaviour or performance of a single core in such a system becomes complicated, as verification struggles to generate inputs and scenarios that model how the rest of the system that interacts with that core.

4.3 Hardware/software co-verification

Probably the most crucial step in embedded system design is the integration of hardware and software. At some point during the project, the newly coded software will meet the newly designed hardware. Arguably, the sooner the better. TLM can enable this co-verification [20], which can also be seen as true co-design. A typical use case is to replace detailed RTL models of processors for higher-level instruction set simulators.

Issues naturally arise when hardware and software are combined. Even if the software was tested and the hardware was verified, validating both at the same time is not a given. The focus of hardware/software co-verification can be either functional (to find bugs) or performance oriented.

Another issue that arises during hardware/software co-verification is the interactions of IPs with the remainder of the system. These IPs are usually developed by third-party companies and may be black boxes. This lack of controllability and observability of the IP can make performance goals hard to reach and verify.

4.4 Power and performance verification

Performance verification is deeply related to how software and hardware interact. However, low-power techniques also have to be taken into account since modern many-core based SoCs have multiple schemes for reducing power, such as dynamic voltage and frequency scaling, as well as selectively shutting off portions of a system.

The power verification challenge is to prove that the system remains operating in a functionally correct way, but also respecting the expected performance, when these schemes are operating. The possibility of having different operating modes for different modules of a system clearly increases the verification complexity, as the same module (and its interactions) has to be verified in ON and OFF states, and others as well.

4.5 Predictability, boundability, and real-time

The bottom line challenge of using a complex system is how to maximally exploit the capabilities of the hardware without jeopardising the system's dependability, verifiability and finally certifiability - the latter being of fundamental importance for high dependability applications.

Many of the advances in the design of processors were obtained by improving the 'average case'. A simple example is the use of pipelines: one single instruction might take more time to be fully issued, but the throughput of the average case will surpass this initial drawback. Similar scenarios are true for cache memories, speculative execution and so on. All these techniques make the system less predictable (in terms of execution time of a given task). If full predictability cannot be reached, perhaps boundability can be achieved, i.e., a given task is guaranteed to execute in a timestep. These two characteristics are important for real time systems, and therefore verifying these characteristics is clearly of interest for MCS. Nevertheless, these are very hard to verify as this a case of the well-known problem of worst case execution time [21].

Alternative ways to build cores and SoCs that are more predictable do exist [21], which can help to meet hard real time constraints.

4.6 Reconfigurability

Space applications are also an example of MCS, and these applications are known to often use FPGA technology. One feature currently present in FPGAs is partial reconfiguration, which implies that the system can effectively change during the mission time.

Reconfiguration could be motivated by reliability reasons (such as to avoid or replace a certain faulty unit) and therefore will keep the intended functionalities of the unit and system untouched. This means that the 'new unit' does not need to be functionally verified again. However, if the system changes into something different, a new functionality is introduced, which naturally requires some extra verification effort, even if it is confined to a component level. Effectively, new verification corner cases are created every time a new unit is introduced into a system.

On top of that, the mechanism that handles partial reconfiguration and its behaviour has also to be verified. If the system has firm guaranties that have to be delivered, such as a certain bandwidth or timing deadlines, these might have to be respected even when undergoing partial reconfiguration.

4.7 High dependability applications and fault mechanisms

Many of the target applications within EMC2 are high dependability applications, which naturally imply special reliability needs. Much of those concerns affect the design of hardware, but some eventually have to be considered during verification as well. For instance, a certain design can operate in a given mode during normal circumstances and, in the presence of a fault, the design shifts to a safe operating mode. The question that remains to be answered is how to verify that the system does the mentioned shift correctly.

The implication of verifying the shift in operating modes is that verification has to be able to 'cause' a fault to trigger the associated fault mechanism. For that purpose, fault injection can be used. A faulty behaviour can be created in simulation, emulation, or even with corrupted software. Currently used simulators normally offer fault injection capabilities through force statements that modify the value of a signal during simulation. The remaining challenge is how to combine that with traditional testbenches and their goals. The workload used for general verification isn't necessarily the same used for assessing the reliability of a system (poor stimuli can mask the fault or make it silent, therefore not reaching the desired fault-mode that is intended).

There are also performance issues that have to be verified when under the presence of a fault. There might be restrictions on how much time the system can take to respond to a fault detection. This time has to be measured and verified. A system can also enter a reduced performance mode when a fault occurs. The performance behaviour under a fault is an interesting verification corner case.

Simulation with fault injection is a time consuming task simply because the number of fault sites grows with circuit size. Formal verification approaches that do not suffer from long execution times can be used instead. The focus on the properties being written and checked changes from functional properties to properties that specify hardware safety mechanisms.

For instance, a design redundancy technique can be used to detect (soft) errors in registers. Properties can be used to specify the ‘safe’ behaviour of these registers. Each property specifies a cause-effect relation, i.e. a logical implication. If a fault occurs, then a reaction must follow. Instead of writing one property for each protected bit, formal verification can use a global property that allows to prove all bits are ‘safe’ in one single run.

5. Existing methods in use at different levels

The following table is a summary of how the existing methods and techniques can be used in different scenarios. It should be read as follows: the upper row shows different levels to which verification is applied. Complexity rises from left to right.

The left hand column lists the different techniques. Some techniques are perfectly adequate for a given scenario, and in that case the corresponding cell says **Y** (yes). The opposite is also true, some techniques simply do not work well in a given scenario. In that case, the corresponding cell says **N** (no). Of course, there is a lot of room for discussion on what an adequate technique is. When a technique can be used in a certain level, but likely restrictions do apply, the corresponding cell is shown as **L** (limited).

The listed levels are block, chip, system, and an MCS with many cores. The boundaries between the levels are not specified. But it should be clear that a block-level verification is a lower effort verification, that could even be (partially) performed by the designer of the module. This is no longer true when moving up to chip- or system-level, which will likely require more effort.

Block-level verification is primarily concerned with the functionalities of a standalone module. Chip-level verification is concerned with the integration of those modules, i.e., focusing on the interfaces and interactions. Finally, system-level verification brings software concerns into play. The last level – MCS with many cores – represents a system that is more complex from a size point of view as well as from a number of features point of view.

Table 1: Methods and usage in different scenarios

Technique / Level	Block	Chip	System	MCS w/ many cores
Code coverage	Y	L	N	N
Functional coverage	Y	L	N	N
Formal methods	Y	N	N	N
Formal apps	Y	Y	N	N
Assertions	Y	Y	L	L
Fully random generation	L	N	N	N
Constrained random generation	Y	L	N	N
TLM	L	Y	L	L

6. Future work

UoBR will use its experience in specification, formal methods, and simulation-based verification approaches to develop and assess verification methods applicable to the designs defined in different EMC2 subtasks (namely T4.1, T4.2, and T4.3).

TVS will be developing and evaluating hardware techniques that enable multi-core processors to execute applications with mixed criticalities.

UTIA will keep working on its FPGA-based system that includes heterogeneous cores (ARM and MicroBlaze cores) as well as accelerators of specific purpose. Verification of this system is already conducted by different approaches, including simulation, co-simulation, and co-verification. A challenge that will be investigated is how to combine stimulus generation at different levels and by different tools (e.g., Simulink, Matlab, C code) with the state-of-the-art FPGA tools like Vivado.

POLITO will work to develop a simulation-based verification environment allowing both functional verification and fault injection. A prototypical virtual platform for the processor architecture will be developed; the platform will allow software execution, and it will offer the capability of injecting transient and permanent faults during simulations, and to compare fault free software execution with software execution in presence of faults.

At NXP we have made the experience that we are successful with Matlab for PHY and DSP code and FPGA for RTL verification. In general System C is not a good fit for our purposes and applications. The next step is to combine the verification models and methods developed into an advanced system design methodology and its integrated design flow. In the coming next months, NXP will work on its software defined radio system starting with the FPGA development. Software and new hardware instructions are developed in parallel to verify the system and its hardware-software co-design according to the requirements.

IFAG will keep working on a formal verification approach that strives to give evidence that safety goals are met by installed hardware safety mechanisms. Based on this concept, extensive infrastructure is going to be developed, namely by using advanced proof automation procedures based on Onespin features and utilities.

TASE have contributed to the requirements gathering process through their expertise in space and avionics applications. These high-level requirements were taken into account when the verification challenges listed in this document were elaborated.

7. References

[1] PIZIALI, A. **Functional Verification Coverage Measurement and Analysis**. Kluwer Academic, 2004.

[2] BERGERON, J. **Writing Testbenches: Functional Verification of HDL models**. Kluwer Academic, 2003.

[3] VSI Alliance. **Specification for VC/SoC Functional Verification**. 2004.

[4] Alan Burns and Robert I. Davis. **Mixed Criticality Systems - A Review,** 2014

[5] KITCHEN, N. and KUEHLMANN, A. **Stimulus generation for constrained random simulation.** IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2007.

[6] CHEN, K.-C. **Assertion-based verification for SoC designs.** INTERNATIONAL CONFERENCE ON ASIC. 2003.

[7] MICZO, A**. Digital logic testing and simulation.** Harper & Row Publishers, 1986.

[8] LACHISH, O. et al. **Hole analysis for functional coverage data.** DESIGN AUTOMATION CONFERENCE, 2002.

[9] Cadence Design Systems and Mentor Graphics. **Open Verification Methodology User Guide**, 2008.

[10] John Harrison. **Floating-Point Verification,** International Symposium of Formal Methods, 2005.

[11] COHEN, O. et al. **Designers Work Less with Quality Formal Equivalence Checking**. Design and Verification Conference, 2010.

[12] Ping Yeung; Larsen, K. **Practical Assertion-based Formal Verification for SoC Designs,** International Symposium on System-on-Chip, 2005.

[13] ABADIR et al. **Formal Verification Successes at Motorola**. Formal Methods in System Design, 2003.

[14] International Roadmap for Semiconductors. **Design Chapter**, 2011

[15] Cadence. **Clock Domain Crossing**. Whitepaper. [Online]. Available: http://w2.cadence.com/whitepapers/cdc-wp.pdf

[16] Mentor Graphics. **Five Steps to Quality CDC Verification**. Whitepaper. [Online]. Available: http://www.mentor.com/products/fv/techpubs/mentorpaper_32969.cfm

[17] FINE, S.; ZIV, A. **Coverage directed test generation for functional verification using Bayesian networks**. Design Automation Conference, 2003.

[18] **RTCA/DO-254 - Design Assurance Guidance For Airborne Electronic Hardware**, RTCA Inc. ,2000.

[19] **IEEE Standard for SystemVerilog** - Unified Hardware Design, Specification, and Verification Language, 2009.

[20] Mentor Graphics, **ZeBu Transaction-based Emulation Solutions.**

[21] Wilhelm, R et al. **Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems**. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009

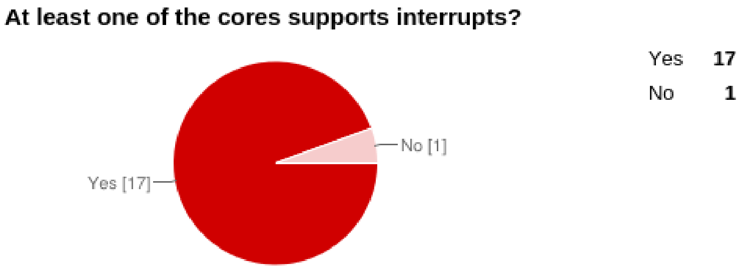
8. Appendix A: Architecture survey

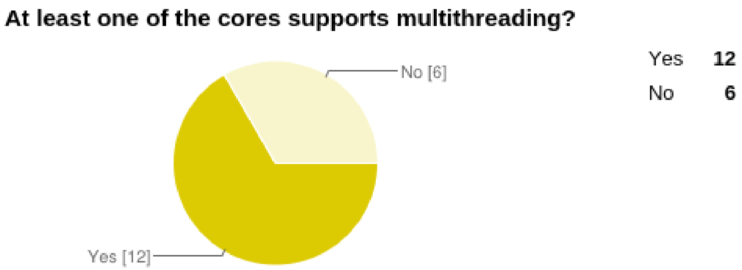
An architecture survey was conducted amongst EMC2 partners. This was an attempt to capture interesting aspects of the architectures being used that could potentially lead to an increased verification effort.

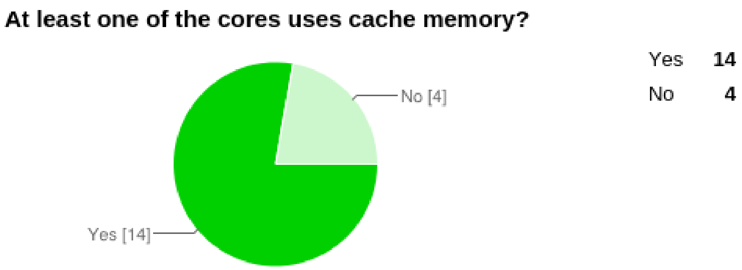
The two-part survey has core-related and platform-related questions. On the cores side, the trend is that most partners will be using cores that:

- are varied, from ARMs to Microblazes, from powerPCs to customised solutions.
- support interrupts and multithreading
- use cache memories extensively
- use pipelines
- support out-of-order execution and (dynamic) branch prediction

The image below shows some of these trends.





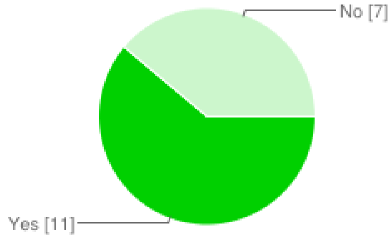


On the platform side, the trend is that most partners will be using platforms that:

- support some level of reconfiguration
- support dynamic voltage and frequency scaling
- offer global time services
- make use of memory management units, likely to include containment features and TLBs

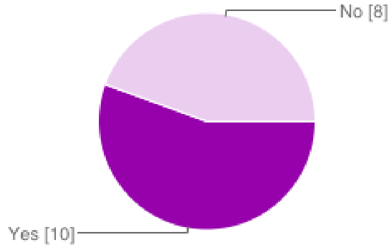
The image below shows some of these trends.

Any dynamic power control/scaling?



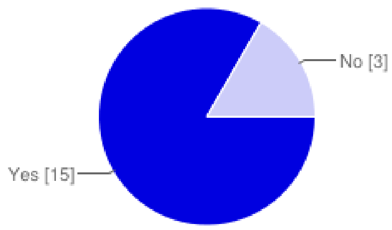
Yes	**11**
No	**7**

Any dynamic frequency control/scaling?



Yes	**10**
No	**8**

Support for global time service?



Yes	**15**
No	**3**