# Early Stage Interference Checking for Automatic Design Space Exploration of Mixed Critical Systems

Emanuele Vitali Gianluca Palermo
Politecnico di Milano - Dipartimento di Elettronica, Informazione e Bioingegneria
Email: {name.surname}@polimi.it

***Abstract*—Significant improvements have been made to support the design of mixed-critical systems by developing predictable computing platforms and mechanisms for temporal and spatial segregation between applications of different criticalities sharing the same computing platform. However the design of such Multi-Processor System-on-Chips (MPSoCs) supporting mixed-critical applications needs methodologies and tools to improve the analyzability regarding system configuration and application mapping.**

**Among the required techniques, in this work we focus on the possibility to identify, at the early stages of the design, possible unexpected interactions among tasks relying to different criticalities. In particular, we introduce a dependency check tool to automatically find possible interactions between tasks during the design of a mixed critical embedded system. The tool searches on an abstract system model for the possible interactions, thus helping pruning all the design configurations not respecting the considered criticality constraints. In this way the methodology can be used to speeds-up the following design space exploration phase based on functional models (e.g. simulation based) avoiding costly evaluations.**

## I. Introduction

Since the early 2000s the design of Systems on Chip (SoC) has switched from custom design to platform based design, trying to reuse as much as possible already existing components in the creation of new systems [1]. As the technology progressed the approach has become more complex, since more and more components are used in a single SoC and many SoC are also connected together using a Networks on Chip (NoC) approach. Nowadays, this increment in complexity started impacting also areas where before the platform based design was not yet considered, or considered only marginally. In such scenario, safety- and mission critical services that previously have been running on dedicated and often custom designed hardware/software platforms are now tightly connected or even execute on devices made of generic hardware/software platforms that have previously been used with non-safety/mission-critical applications only. The interaction between them is not always easy to predict, and two main approaches are used for the chip design: simulation and formal methods. They are often mixed, as in [2] in order to use the best features of both the approaches (completeness of the simulations and fast design space exploration of the formal methods). Since now the components are tightly connected, adding new components for new functionalities can create undesired behaviors [3]. Moreover having only dedicated components leads to the under-use of some resources, thus creating a waste of area, energy and increasing design costs.

To solve these problems, the mixed criticality approach has been proposed. With this methodology more functionalities are integrated on a common platform. The obvious drawback is that it is difficult to predict all the possible interactions between functionalities during the design phase and often those only emerge during the late design or validation phase. For example the use of a common memory is not taken in consideration when designing tasks, because they are planned to work on dedicated hardware and only in a second phase are integrated in a common platform. Another possible source of hidden dependencies is the abstraction: layered models of the SoC are introduced to support the design of different components by different teams [4]. A layered model is a model of an entire SoC composed by a number of models, where every one of these models takes care of a part of the design, at a different level. As an example the application software abstraction, without considering hardware platform details, can be considered a layer, the actual design of the hardware platform resources is another layer, the physical/geometrical abstraction (in terms of different chips, or boards and their placement) is another one and so on. This is done to let every team focus on the details of a part of the design, leaving all the other pieces to other teams with different expertise. Usually the division of the work is by means of horizontal slices, considering the different layers, however it is also possible to split the work at the same level. As an example at the application-software level a team will only cure the design of the critical part of the SoC, while another team will focus more on the non critical part. The model of a single layer usually makes some assumptions on the other layers and some dependencies may be not considered. For example if some OSs have different memory addressing space, there is no dependency found from a logical point of view. But if the memory is only one physical component, even if there is no logical dependence there is a physical dependence, that is not visible in the layer above since it only consider the logical perspective. The abstraction of the layers is needed in order to split the work between different teams and to try to separate some issues on different layers, but on the other hand it can introduce some hidden dependencies that can undermine safety.

In the mixed-criticality context the design paradigms used are the previously cited simulation and formal methods [5],[6] but even mixing them the design process is long, because of the huge space possibility. Moreover a design in the mixed-critical context has to guarantee that the low priority tasks cannot interfere with the critical ones: to do this methods like Fault-Tree Analysis (FTA, [7]) or Failure Modes and Effects Analysis (FMEA, [8]) have to be used. They are used to evaluate system reliability when the design is almost finished and are hard to maintain, in particular if the design is done in different distributed teams. Other methods have been suggested

This work was supported in part by the EC under the grant CONTREX FP7-611146.

to check reliability in an early phase of the design. An example is the work in [9]. The authors suggest to check dependencies at software component level: every software component has to list all the components that it directly uses and with this information generate the system wide software component dependency relations. No cross layer issues are considered, and everything is evaluated at software level. Another example is the AADL language [10] error model annex, that permits to build a FMEA in the design phase.

The proposed solution is a methodology to performs a rapid exploration of the possible design alternatives considering all the layers. The objective is not to find the best solution but to rapidly prune the design space from the set of solutions that clearly do not satisfy some constraints, thus speeding up the following detailed evaluation. The methodology consists in performing an interference check on the whole system, and only the possible configurations where the functionalities with high level of criticality are not disturbed by the ones with lower criticality will be selected, thus pruning all the others. That selection is used as starting point for further refinement in the evaluation [5][6][11].

The idea of collapsing all the models in an unique model to search for dependencies in depth has already been proposed by [12]: all models are united in a graph and there are two possible interaction between them, modeled as arcs: model dependencies (i.e. dependencies that are present in the original model and are reported in the global one) and cross-layer dependencies (created in the global model when a model of a layer is mapped on a model of the layer below). Then a Breadth First Search is performed and if a path is found, some controls are needed to check if it is a real dependence or not. The model can be modified to eliminate the dependence. Their methodology can perform validation: given the model they perform iterative exploration (if a path is found,the model can be modified adding some constraints and the process is restarted).

The proposed solution has a different perspective: while the solution proposed in [12] is used vertically in the design flow, the suggested methodology is used only to rapidly check if in the input model (that is the system graph) there is an interference or not. The proposed solution is included in a framework for automatized Design Space Exploration (DSE), and in particular the current methodology is used to quickly prune the design space, focusing the subsequent accurate analysis of the solutions (typically simulation-based) only for those that have been considered possibly valid. This different approach can be seen in Figure 1, where the black path is the proposed approach, while the red one is the usual approach to the design. Both starts with the composition of a complete layered model of the system including design alternatives. The main difference is that while the usual approach is an iterative work completely handled by the designer including the validation of feasible design alternatives, the proposed solution removes the human intervention. It adopts an automatic brute-force approach to filter-out the solutions that clearly are not satisfying the non-interference rules, thus limiting the number of sub-sequent evaluations to be done. Despite the large set of evaluation needed by the interference checker, the running time needed by the proposed approach is kept limited.

The focus of this paper is the core element of the methodology: the interference checker. While in Section II the input system model, the model-to-model transformation and the actual search for interference have been presented,

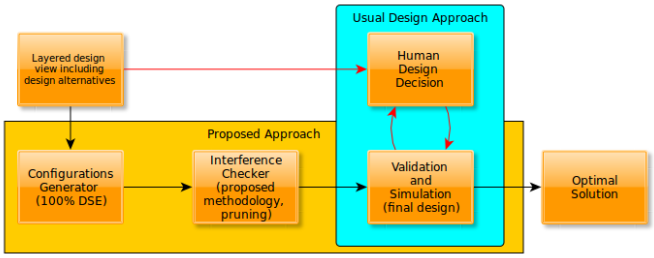



Figure 1. Complete framework structure

Section III presents some validation results on top of two case studies of different complexities. An entire example of fast pruning adopting the proposed approach has been also shown in Section III, while Section IV concludes the paper.

## II. Methodology

The proposed methodology is a search of the possible interactions between two functionalities or tasks at different level of criticality in the whole system architecture.

Since the design of the system is divided among teams and different models, it is simple to capture and handle issues at a given abstraction level, while it is difficult to capture cross-layer dependencies. For example timing across the whole platform cannot be completely predicted while modeling Operative Systems (OS) only, because it may happen that a resource is shared among multiple OSs running on the same platform. In order to search the interferences, a model to model transformation is performed and all the models of the different layers are merged. The analysis is then performed on the resulting aggregated model by checking if a *source* component can interfere with a *target* component. If an interference is found, for sure the *source* can somehow interfere with the *target*, while it is not true the opposite. If no interference is found, it is still possible that the configuration is not viable since not all the low level design parameters are considered at this level (e.g. two connected TDMA buses can guarantee timing-independence, however the time-slots have to be properly allocated or different tasks have been allocated on the same processor, however there is not enough computing power to satisfy the performance constraints). However, the methodology is highly useful because it can be used to quickly prune the design space in the early stage exploration phase. The proposed methodology skips all the low-level validation checks leaving them to the following stages of the design.

### A. System Model

The layered design paradigm is adopted as input. To be able to perform the search on the whole system, it is mandatory to have a single input model (from now on called input). The input can be obtained merging different models of already existing components or made by different design teams, what is important is that it abide to the requirements described below. It is a graph on five levels, where the nodes of the graph are the components of the SoC and the edges are the connections between components. The components in the input have to be assigned on one of the following five layers:

- *Layer 1: Functionality.* This layer refers to the pure functional view of the system, by including the lists of functionalities.

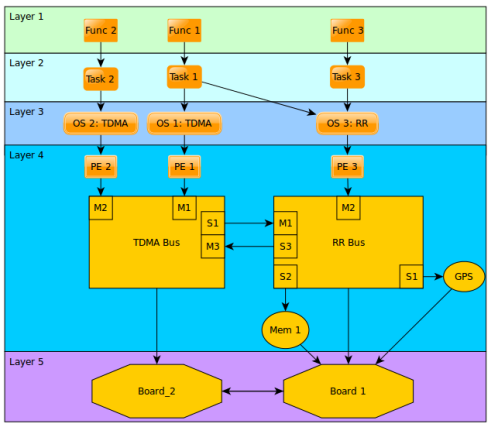



Figure 2. Input examples

- *Layer 2: Tasks.* This layer includes the software elements (processes, threads) implementing the requested functionalities.
- *Layer 3: Controllers.* This layer includes the system software controllers, with their characteristics (e.g. Operative Systems)
- *Layer 4: Resources.* This layer refers to the hardware architectural view, including components such as processing units, memories and buses.
- *Layer 5: Physical.* This layer refers to the geometrical view in terms of physical elements such as chips and/or boards.

An example is provided in Figure 2. For the proposed methodology to work only a few information are needed for every component and for the connections. Since up to now only the timing interference search has been considered, all the specifications of the components and of the edges are related to this kind of search. It is possible that with the addition of different perspectives more specifications will be required. The idea is to have a global input that is able to work with all the possible perspectives, while internal views will be created for every different perspective to search on.

*1) Components Specifications:* The components are the nodes of the graph. For every node there are two mandatory information: the name (that is used as identifier, must be unique) and the layer they are placed in. Then up to now more specifications have been individuated for the components in layer 3 (OS) and layer 4 (resources).

Layer 3 components have a scheduler item which will contain all the thread or processes that are mapped on that operative system, with their priority, plus a field that declare which priority handling policy that OS will be using if more than one task is mapped on it. Until now the identified priority handling policies are three:

1) Round Robin: everything may interfere with everything that is mapped on that component.
2) Priority: everything has an associated priority value, the higher ones can interfere with lowers.
3) Time Division Multiple Access (TDMA): the tasks cannot interfere with each other.

Layer 4 components requires the list of ports of the component, the type of the component, the priority policy. There are three possible types of port: master, slave and master/slave. In the model the master/slave port has been represented as a pair of ports, where each one contains, among the others data, the id of the other one (the master will have the id of the slave and vice versa). The other data required in the creation of a port are: identifier, priority, and a field with the type of port: master or slave. For this model, master ports are the one that are involved in the arbitration procedures inside the component, while slave ports are just used as exit points from the component that contains them towards other components. Taking as example the Round Robin bus, port M2 is a master port, S1 and S2 are slave ports and port S1/M3 is an example of master/slave port.

*2) Edges Specifications:* Every component has to be connected to the other components of the system and this is done using the edges. There are two types of edges: intra-layer or inter-layer.

The intra-layer edges are a way to represent dependencies among components on the same layer. Intra-layers edges have different meaning depending on the layer where they are placed:

- Layer 1: The edge between two functionalities represents that one of the two calls, or interacts, with the another one.
- Layer 2: The edge between two tasks represents an intrinsic dependency on the another one (e.g. waits for data or synchronization purposes).
- Layer 3: The edge between two OSs determine that one of the two can control the another one (e.g. inner OS).
- Layer 4: The edges at resource levels represent the topological connections between components.
- Layer 5: The edges at physical level means that two components are physically coupled (e.g. there can be thermal interference).

The inter-layer edges are the so called *mapped-on* relations. The *mapped-on* edges are the links between a component of layer $L$ and a component of layer $L+1$. Every component has to be linked with at least one component of the layer below. For example a process has to be scheduled on an Operative System (that can be a Real Time OS, a normal OS or an hardware controller). The OS need to be mapped on a processing unit, and so on.

These edges can be also used as an expression of the design space. If there is only one mapped-on edge for a component to the layer below, this represents a design decision. In case there is more than one mapped-on edge coming from a same component, each one of them represent a possible design alternative and thus a decision not yet taken. The edges $Task1$ to $OS1$ and $OS3$ in Figure 2 are an example of such possibility.

For all edges the required information are the source and the target components. There are two exception to this rule. The first one is for the edges that map a task on an OS; the OS will need to know on which slot the task is scheduled (for managing the priority scheme), so this information has to be added to the edge. The second exception is for the Layer 4 intra-layer edges, the topological connections. Every edge has to include information about the ports in the source and in the target component that are used for that connection.

### B. Interference Check

Once the graph representing the whole system has been built, it is possible to search for possible interferences: from the graph representation point of view this is a search for a path connecting two tasks. The search can be performed at different level depending on the scope the designer want to cover.

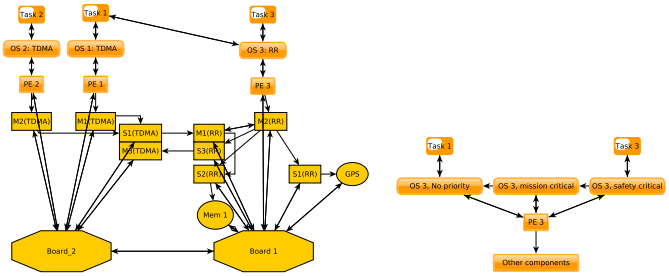
An internal view of the graph has to be created to highlight the critical information for the considered perspective and the search is performed on that view. Example of view are the timing view (with all the possible path between resources), the data view (with the addressing spaces) and so on. As said above, different views have been considered, but for now only the timing view has been completely designed and implemented. For this reason this work will be from now on focused on this single view.

*1) Timing View:* The input model is used to obtain an internal graph where the search can be performed. This operation is done because the einput has to be maintained as general as possible, while the internal representation have to be tailored as needed for the search. The Functionality Layer is not considered in the timing view, since functionalities are included into tasks, and they cannot interfere from a timing point of view. It has been introduced because it has to be inserted in the input, but will be ignored from now on since it is not influencing the timing perspective.

*a) Components Transformations:* To create the view some transformation have to be performed on the input. Components on Layer 1 are removed, with all the related edges. For components on Layer 2 and 5 one node will be created in the view graph. For components on layer 3 and 4 several nodes are created in the view, according to extra information collected in the input model.

*Layer 3:* The priority handling policy is checked for each component at the control layer. If the component is a Round-Robin OS, one node is created in the internal timing graph; if it is a priority OS one node is created for each different priority handled and dependencies between them are created accordingly to the priority level (i.e. intra-layer arcs from higher priority nodes to lower priority nodes). Finally, if the OS is TDMA then one node for every scheduler slot is created, and no dependencies between those nodes is created.

*Layer 4:* The management at resource level is similar to layer 3 for arbitrated components (such as buses), while the same as layer 2 and 5 for the remaining ones (e.g. peripherals and memories). Again the starting point for arbitrated resources is a check on the priority handling policy. For a Round Robin component a node for every port is created. Then all the master ports are connected each other with a bidirectional edge. They are also connected through a mono-directional edge to all the slave ports. For a priority component a number of nodes equal to the different priority level among the master ports is created, while a node for every slave port is also added. Then single direction interference edges are created according to the priority. More in detail, the first edge goes from the highest priority port to the second highest component, then from the second to the third and so on. The lowest priority master is connected to all the slave ports. For a TDMA component a number of nodes equal to the number of ports is created. Every master is connected with a mono-directional edge to all the slaves, but no master is connected to another master.




(a) Example of a timing view, created from the previous input

(b) possible OS loop

Figure 3. Example of interferences

*b) Edges Transformations:* After the input component transformation and internal graph node generation, the original input edges have to be mapped on the internal model, and the *interferes-with* edges have to be created. Those edges are the opposite of the *mapped-on* and symbolize when a component of *layer L* can somehow create a timing interference with a component of *layer L-1*. For example an OS can always interact with a task that is scheduled on it.

The edges are created as follows:

- Layer 2 to layer 2 edge. This edge is replicated as it is in the original input model.
- Layer 2 to layer 3 edge. The target component of this edge is no more the OS, but the scheduler slot associated. This information is included in the description of the edge within the input model. An *interact with* edge is added in the opposite direction.
- Layer 3 to layer 3 edge. The original edge explodes into several edges starting from all the nodes derived by the source component and ending to all the nodes derived by the target component.
- Layer 3 to layer 4 edge. Those edges are transformed into a multi-to-one link where the source nodes are those created by the original component at layer 3, while the target node is the node associated to the layer 4 component. *Interact with* edges are added for every new *mapped on* created in the opposite direction.
- Layer 4 to layer 4 edge. An edge connecting the node associated to correct port of the source and destination component is created. This information is included in the input model.
- Layer 4 to layer 5 edge. All the nodes derived from the original component at layer 4 must be connected to the layer 5 node. *Interact with* edges are added for every new *mapped on* created in the opposite direction.
- Layer 5 to layer 5 edge. This edge is replicated as it is in the original input model.

*2) Search Algorithm:* After the creation of the view the search for possible interferences is performed. This operation is executed on the customized view that has just been created. It is a two phases operation: during the first phase the master task are propagated to the master ports of logical components, while in the second phase the actual search for interferences is done. The first phase is needed to show some dependencies that are not immediately visible from the model, and are explained with an example. Lets suppose that the connection between the bridges in Figure 2 is monodirectional from the TDMA bus to

the RR bus. It may seem that Task 3 cannot interfere with the other two since the requests can only go from the TDMA bus to the RR bus. But there is a sort of "indirect conflict" which is: if T3 is using the RR bus, and a request come from the TDMA bus to the RR bus, this request may be stopped in the next arbitration. And this can possibly create an interference if the signal that comes from a high priority task is stopped and a signal that comes from a lower priority task is selected in the next arbitration.

In order to solve the indirect conflicts the master tasks are introduced: every *master port nodes* has a list of layer 2 (task) items. The members of this list are the tasks that can reach the node from a node that represent a *slave port node* of another component. This means that the master port can receive requests from the source slave node that is the exit point of the previous component, and those requests can generate an arbitration (for example M1 RR receives requests from S1 TDMA). In our example the master of node M1(TDMA) is task 1, M2(TDMA) is task 2, M3(TDMA) are task 1 and task 3, port M1(RR) are task 1 and task 2, M2(RR) are task 3 and task 1.

This list is build performing for every task a search similar to the Depth First Search (DFS) and adding the task as master whenever a new component is found. In Figure 3a an example of a timing view is shown. The search is performed on this graph. It is not a simple DFS because in case of a *master port node* the set of possible next nodes to be considered during the search is limited to the *slave port nodes* only, thus avoiding to have a sequence of two *master port nodes*. (so for example the edge between the nodes M1(RR) and M2(RR) is ignored). It is possible that more than one *master port node* originated from the same component have the same task inside their list: this happens when that task can reach the component from different paths as in Figure 3a where task 1 is master of both master ports of the RR bus. M1(RR) and M2(RR) are reached from different paths, the first comes from the TDMA bus, while the second from the Processing Element 3 (PE3).

Once all the master tasks are assigned the search for interferences can start. Also this search is performed on the timing view, now enriched with the master task lists for every level 4 master node. This search will give a positive response (that is, there is interference) if a path to the target task is found or is found in an *arbitration conflict check*. The *arbitration conflict check* happens every time a search enters in a master node associated to an original arbitrated component to check if all the other reachable master ports include the target task.

The basic algorithm is similar to the one used for the master tasks search: it is still DFS-like, but the master to master edges are now useful. The master to master edges are important because through them the arbitration conflict check is performed: if there is an edge between two *master port nodes* it means that the task coming from the source can interfere with the tasks that are mapped on the target node, because of the arbitration algorithm of the original component. What is impossible to find is a slave to slave edge, because the *slave port nodes* are the exit point of a component and have to be related to an entry point of another component that is a *master port node*. This information has been used to recognize the *master port nodes* that are involved in an arbitration: *slave port node* are used as checkpoints and all the *master port nodes* that are found between two different *slave port nodes* are involved in an arbitration conflict check.

The scope of the search can be limited: it is not mandatory to examine every time all the possible interferences down to the physical layer but can be stopped at the OS level (layer 3) or at logical level (layer 4). To be sure that no interference are found a search must be done at OS level and one at the physical one. This happens because the OS transformation in more nodes in the timing view can create loops when involving the underlying layers. An example of these loops is reported in Figure 3b where a priority OS has been transformed. Given the *interference with* edges between layer 3 and 4, passing through the processing element 3, task 1 can interfere with task 3. This should not happen because the OS will block the execution of task 1 whenever task 3 will require it since task 1 is mapped at no priority level and task 3 at safety critical level. To avoid this when considering level 4 and 5 it is given as hypothesis that there is no interference at OS level, and is forbidden to search paths passing from more than one layer 3 node. It is important to highlight that tasks coming from other OSs will be mapped as master task in the Layer 4 components so this limitation will not lead to loss of information.

Another important feature is that with this model is still possible to leave some design decision suspended: taking for example the previous Figure 3a, Task 1 is mapped on two different OSs. If no interference is found with this input it means that both the configurations are valid, so this decision can be evaluated in a following step. Otherwise there is at least one of the design alternatives not respecting the no-interference rule.

## REFERENCES


[1] H. Chang, *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Springer US, 1999.

[2] S. Künzli, F. Poletti, L. Benini, and L. Thiele, "Combining simulation and formal methods for system-level performance analysis," in *Proc. Design, Automation and Test in Europe (DATE)*, March 2006.

[3] K. H. Braithwaite and J. M. Atlee, "Towards automated detection of feature interactions," in *FIW*. Citeseer, 1994, pp. 36–59.

[4] AUTOSAR: The AUTomotive Open System ARchitecture. [Online]. Available: http://www.autosar.org/ (Last visited on: 2016-03-20).

[5] F. Herrera and I. Sander, "Combining analytical and simulation-based design space exploration for time-critical systems," in *Specification Design Languages (FDL), 2013 Forum on*, Sept 2013, pp. 1–8.

[6] Z. Jia, A. Núñez, T. Bautista, and A. Pimentel, "A two-phase design space exploration strategy for system-level real-time application mapping onto {MPSoC}," *Microprocessors and Microsystems*, vol. 38, no. 1, pp. 9 – 21, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933113001361

[7] "Fault tree analysis," The International Electrotechnical Commission, Tech. Rep. IEC 61025, December 2006.

[8] "Analysis techniques for system reliability - procedure for failure mode and effects analysis," The International Electrotechnical Commission, Tech. Rep. IEC 60812, January 2006.

[9] H. Ding and L. Sha, "Dependency algebra: a tool for designing robust real-time systems," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, Dec 2005, pp. 11 pp.–220.

[10] SAE international: Architecture Analysis and Design Language (AADL). [Online]. Available: http://www.aadl.info (Last visited on: 2016-03-20).

[11] F. Herrera, I. Sander, K. Rosvall, E. Paone, and G. Palermo, "An efficient joint analytical and simulation-based design space exploration flow for predictable multi-core systems," in *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2015, pp. 2:1–2:8.

[12] M. Möstl and R. Ernst, "Handling complex dependencies in system design," in *Proc. Design, Automation and Test in Europe (DATE)*, Mar 2016.