Towards Creating Flexible Tool Chains for the Design and Analysis of Multi-Core Systems*†

Philipp Reinkemeier[1], Heinz Hille[2], and Stefan Henkler[1]

[1]philipp.reinkemeier,stefan.henkler@offis.de
[2]heinz.h.hille@daimler.com

Abstract: With the ever increasing complexity of today's embedded systems, also the complexity of the design and development processes tends to grow. Experts from different domains and/or organizations work on different aspects of a product. Tool support is necessary for these activities, like design of hardware and software, requirements engineering and verification. While an increasing number of tools is available, these tools are usually not integrated. Hence, it is difficult to establish a traceability between data created by these tools and to access data created by a particular tool from within another tool that needs this data. As a consequence, the same information is manually created multiple times in different tools, or ad hoc tool integrations are created, which often work only for a particular use case and specific tools.

In this paper we propose a strategy for flexible tool integration, which is to a large extent independent from concrete tools to be integrated. This strategy is based upon a metamodel defining common semantics of data, which is created and shared between the different tools involved in a development process. The metamodel is an extension of a framework developed in the SPES2020 project with added concepts necessary for the design and analysis of multi-core systems.

1 Introduction

Due to the complexity of today's embedded systems, their development heavily relies on tool support. While an increasing number of tools is available, these tools are usually not integrated. Hence, it is difficult to establish a tool chain, where data created by a particular tool is accessed from within another tool that needs this data. As a consequence, the same information is manually created multiple times in different tools, or ad hoc tool integrations are created, which often only work for a particular use case and specific tools. The unsynchronized tools cause gaps in the needed workflow and the project specific tool integration is error prone. This increases the development time and costs of the product.

An important requirement for managing complex system development with potentially participating engineers from different disciplines is the support of traceability between ar-

*This work was supported by the Federal Ministry for Education and Research (BMBF) under support code 01IS11035M, 'Automotive, Railway and Avionics Multicore Systems (ARAMiS)'.

tifacts. This is not given per se as the integration is typically done only on the activity (process) level. The ISO 26262 [Int] defines traceability for each source of a safety requirement at the higher hierarchical level or for a lower level if the safety requirement is derived. SPICE [Gro07] defines in ENG.6.BP9 that the consistency and (bilateral) traceability of software requirements should be guaranteed to software units.

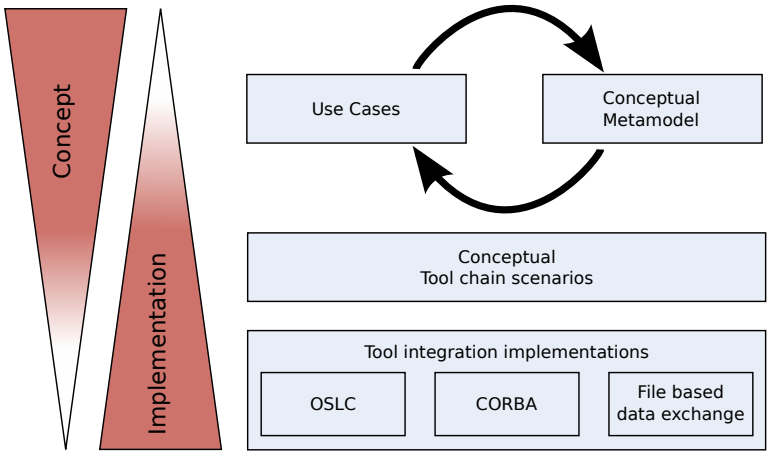


Figure 1: Overview of the tool chain definition and implementation strategy

To overcome these limitations, we propose a layered strategy for the creation of flexible tool chains. Figure 1 shows an overview of this strategy consisting of three layers. The top most layer forms the conceptual basis for every tool chain. Based on the steps of the targeted development process, use cases are derived describing the needed methods supporting the development steps. These use cases also define necessary input and output information for the methods. In turn these information and relationships between them, are captured by means of a metamodel. The goal of the metamodel is to describe a common vocabulary for the use cases and their described methods. In the middle layer we define tool chain scenarios coupling multiple methods from the use cases identified before. This layer still ignores particular tool instances or the way data exchange between tools is implemented. Instead the chains refer to classes of tools, like a requirements management tool. The exchanged information are described in terms of concepts of the metamodel. Manifestations of the conceptual tool chains are located at the bottom layer. Here tool classes are replaced with a particular tool candidate, e.g. requirements management tool becomes Rational DOORS. Further, a particular integration strategy is selected for each connected pair of tools. For example tools can be integrated by means of webservices as suggested by OSLC [JS11] or a simple XML format can be defined or reused that can be imported and exported by the considered pair of tools. This strategy allows us to replace a particular tool instance without changing our conceptual tool chain we had in mind and it is technology independent.

In the German ARAMiS (Automotive, Railway and Avionics Multicore Systems) research project we are currently following this strategy in order to build tool chains for the design

and analysis of multi-core systems. The goal of the ARAMiS project is to enable the usage of multi-core technologies in embedded systems, focussing on the automotive, avionics and railway domains. The challenges include, but are not limited to, efficient utilization of the cores, and coping with undesired interferences between functions due to the increased parallelism of an underlying multi-core platform. So in the project methods and tools are being developed and/or extended, addressing these challenges. As a starting point for the top most layer of the strategy, we investigated a modeling framework that has been developed during the SPES2020 project (see [BDH+12]).

This framework is refined by a metamodel with a well defined syntax and semantics and which already covers a broad set of use cases as described in [DHH+11]. That metamodel has its roots in the metamodel for *Heterogeneous Rich Components* (HRC) [Par09], developed in the SPEEDS project. While we analyzed the SPES2020 metamodel, it turned out that its provided concepts are not detailed enough to capture the use cases considered in the ARAMiS project. Thus, we developed an extension of the metamodel that we present in this paper. Further, we present a particular use case and tool chain scenario and discuss how this scenario can be realized based on the extended metamodel. The implementation of the tool integration, which is the bottom layer, is out of scope of this paper and subject to future work.

The rest of this paper is structued as follows: In Section 2, we briefly describe the basic concepts of the SPES2020 framework, as it is the basis of our proposed extension. Section 3 elaborates the proposed metamodel extensions. In Section 4 we discuss a tool integration scenario backed by the metamodel. Section 5 concludes the paper and highlights future work.

2 Basic Concepts

The metamodel we propose is an extension of the modeling framework developed in the SPES2020 project [BDH+12] with its refined syntax and semantics as defined in [DHH+11]. We especially focus on extensions necessary for the design and analysis of multi-core systems. The SPES2020 metamodel for architectures of embedded systems provides syntax and formally defined semantics of its artifacts and requirements. Using the metamodel, architecture models and requirements of embedded systems can be captured, and its semantics provide means to reason about such architecture models in a compositional way based on the requirements.

A key concept of the modeling framework is to structure models establishing different views of the developed system according to a two-dimensional schema like shown in Figure 2. This schema identifies a class of viewpoints in the architecture metamodel as shown at the bottom of the figure. These viewpoints are typically found in the design processes of complex embedded systems like in the avionics or automotive domain. The viewpoints are complemented by abstraction levels as often systems are modeled with an increasing level of detail and refined towards an implementation.

Alongside the two-dimensional design-space, the metamodel is based on a component con-

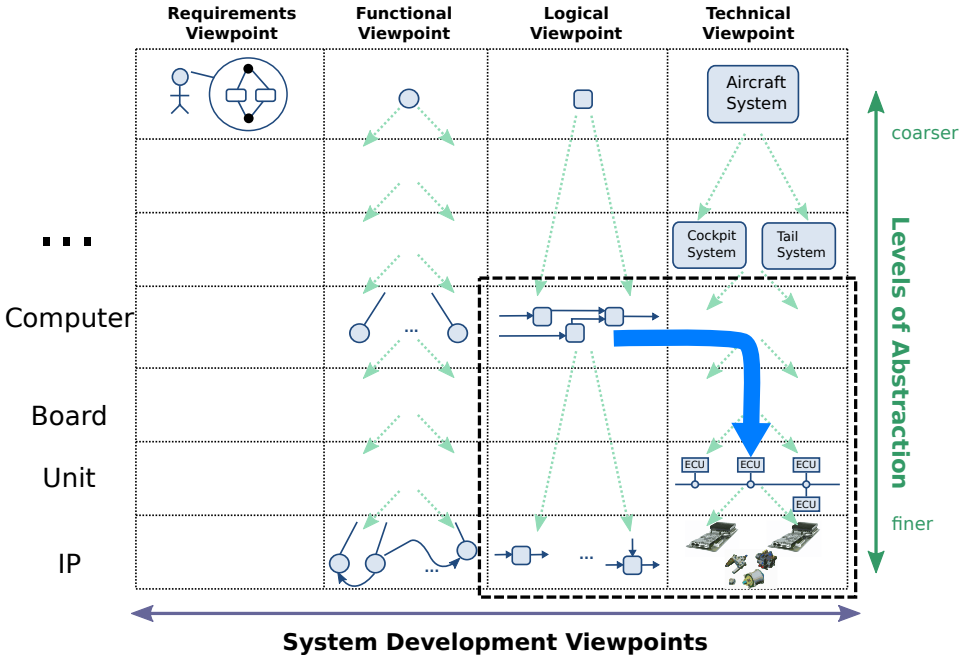


Figure 2: Matrix of abstraction levels and perspectives

cept similar to those found in e.g. SysML [Obj08], EAST-ADL [ATE08] and AUTOSAR [AUT09]. A component has a well defined syntactical interface in terms of its ports. Components can be assembled forming so called compositions, which are again regarded as components. A specialty of this component notion is its strict separation of specification of different behavioral aspects and a realization of these aspects. Aspects specifications are assumed to be formulated using contracts promoting an assume/guarantee style. That means a contract explicates assumptions about the environmental context of a component under which its it behaves as guaranteed by the contract. Figure 3 depicts these main features of the component notion.

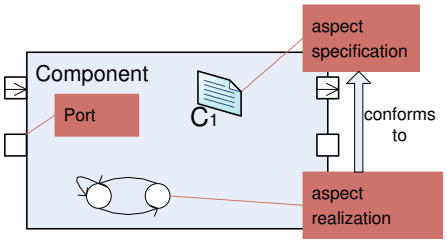


Figure 3: Component concept, aspect specification and realization

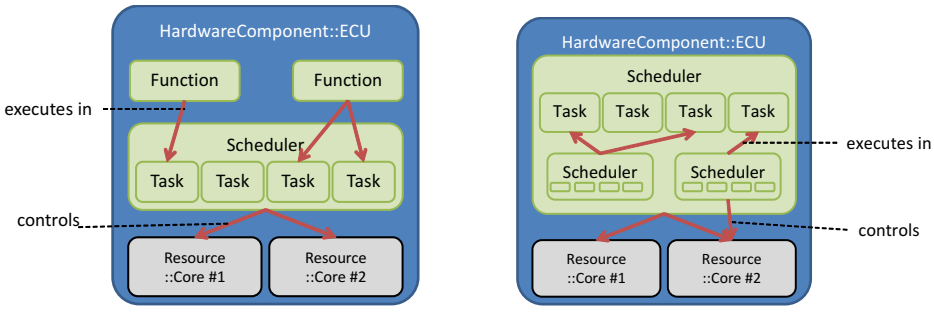
3 Multi-Core aware Technical Architecture Model

Complexity of safety-critical systems is ever increasing as the number of functions realized by these systems keeps growing. Moreover, an increasing number of functions is realized in software, which are then integrated on the same target platform in order to save costs. Hence we observe a paradigm shift from rather loosely coupled federated architectures, where each subsystem is deployed on a dedicated hardware node, to integrated architectures with different subsystems sharing hardware nodes.

This leads to a decrease in the number of hardware nodes in an architecture, and such systems are inherently more complex and difficult to design for a couple of reasons: In federated architectures the different subsystems are independent to a large degree (only the communication infrastructure between hardware nodes is shared). Thus, realizing and verifying a segregation of different functions in the time and space domain is relatively easy for such architectures, which is a prerequisite for enabling certification or qualification according to respective safety standards (e.g. ISO26262).

In contrast, in integrated architectures with multiple subsystems sharing hardware nodes, this independence is compromised. On the one hand, this leads to safety related implications as the potential of error propagation from one subsystem to another subsystem increases. On the other hand, the compromised independence increases complexity of verification and validation activities, as subsystems now influence each other in a potentially undesired way due to their resource sharing. Using multi-core target platforms, the design and verification of such systems becomes even more challenging. This is due to an increased interference between functions resulting from more shared resources like e.g. on-chip-buses and memory shared by the different cores and hence the functions executing on these cores. Thus, a verification of the timing behavior of multi-core systems is of paramount importance in order to analyse the interference due to resource sharing.

To support the exchange of data provided and consumed by timing analysis tools, we developed an extension of the SPES2020 framework [BDH+12, DHH+11], adding concepts to represent a *task network* deployed on a multi-core architecture. These concepts are embedded in the already existing component notion described in Section 2.



(a) Symmetric Multiprocessing

(b) Hierarchical scheduling with core affinity

Figure 4: Scheduling model of a technical architecture

Basically this extension consists of three fundamental concepts:

- *Functions* define the atomic functional units to be executed on a target platform.
- *Schedulers* own a set of *tasks*, each of which executes some of the *functions* or a child scheduler.
- *Resources* represent the communication or computation resources provided by a component. The number of resources owned by a component denotes the available degree of parallelism when multiple functions are deployed on the same component.

Invocations of functions (execution requests) are modelled by events occurring at particular ports, and the same holds for completions of functions. Dependencies between functions are modelled by connecting input/output pairs of ports, unifying behavior observable at these connected ports. Functions without dependencies are connected to event sources that represent assumptions about the evironment. For each event source, we define an activation pattern that characterizes event occurrences. Activation patterns can be described that conform to the formalism of event streams, which is well-known in the scheduling analysis community (cf. [Ric05]).

The resources provided by a component are shared by all functions deployed on the component, and therefore access to these resources must be scheduled. This scheduling is modelled by a *Scheduler* entity that has important attributes such as the choosen scheduling strategy (e.g. fixed priority based). As suggest by Figure 4, a scheduler controls a set of resources, which might be the different cores of a multi-core ECU. The set of *tasks* owned by a scheduler indirectly represent the functions that are executed on the component, using its resources. Each task has scheduling-specific attributes, like the order of Functions executed in its context or a priority assigned to it.

With regard to scheduling execution of functions on multiple cores, the model allows to describe symmetric multiprocessing scenarios like depicted in Figure 4a and also scenarios with fixed assignment of functions to cores by means of a hierarchical scheduling like depicted in Figure 4b. In the former case it is assumed that it does not matter on which core a function is executed. The activated functions are simply dispatched on the available resources/cores. In the latter case a child scheduler can be modelled, which is executed within the context of a task of the top-level scheduler. A relation (labelled *controls* in Figure 4b) allows to specify a subset of the resources/cores used by the tasks of the child scheduler.

In [BDH+12] and [DHH+11] concepts have been proposed to capture a model of such a technical view of a system. However, these concepts neither allow expressing symmetric multiprocessing scenarios, nor a binding of a child scheduler to a subset of resources/cores.

4 Tool Integration Use Case

As data races may destroy needed data or introduce non-determinism in the execution of a program, they should be detected and avoided. However, it is known that statically

detecting race conditions in programs using multiple semaphores is NP-hard [NM90]. So exhaustively detecting all data races is infeasible. Still heuristic algorithms can be used to find *potential* race conditions. For example the Bauhaus tool suite employs this strategy. Unfortunately, data races detected that way may not necessarily occur in the program when it is deployed on a target platform. That is because execution orderings of some instructions assumed in the data race analysis, potentially leading to a race condition, cannot happen in the deployed program. This is because data race analysis does usually not consider the time needed to execute a *Function*, nor the strategy used to schedule them. Consequently, the analysis must assume that functions can be executed concurrently, while in reality they cannot.

To reduce the number of false positives of detected data races, the analysis can be extended by considering additional information whether two accesses to a shared resource can be executed concurrently. With regard to the *task network* model presented in Section 3, this question reduces to the simpler question whether two *Functions* can be executed concurrently. Assuming for example a single core platform and a non-preemptive scheduling strategy, accesses to a shared resource can never be executed concurrently as the task executing the function must relinquish the CPU. Of course this is not necessarily true for a multi-core platform, where multiple task may be executed in parallel.

While for example Bauhaus supports the manual definition of the timing behavior of the analyzed program, this process is error prone. In addition the very same information is already computed and available in timing and scheduling analysis tools. For example the tool SymTA/S can calculate response time intervals for all functions of a task network. Combined with information about activation dependencies between functions (and therefore tasks), one can deduct whether functions can be executed concurrently.

Thus a timing and scheduling analysis tool can provide valuable information to a data race analysis tool in order to reduce the number of false positives in the detected data races. In the easiest case, this provided information is a binary relation on the set of functions of a program. For every pair of functions in that relation, it is must be guaranteed that they cannot be executed concurrently.

These findings led us to augment the concepts in the metamodel extension, as presented in Section 3, with a relation expressing whether a set of functions can be executed concurrently. The class diagram in Figure 5 shows the relevant excerpt of the metamodel, with the meta class `NonConcurrentExecution` representing this relation. Based on these metamodel concepts, we can now address the dicussed use case and derive a tool chain scenario. Figure 6 gives a graphical overview of this tool chain. First a timing model must be created as input for a timing and scheduling analysis tool, for example SymTA/S. Conceptually, the information contained in this timing model is covered by the metamodel concepts discussed in Section 3. That means the entry point of the tool chain can be linked with other tools providing this data. For example SymTA/S is able to import AUTOSAR models. Based on this timing model, charateristics like response time intervals for the individual functions are computed by the timing analysis tool.

In addition the information about concurrent execution of functions is provided (the box labelled *Concurrency Information* in Figure 6). This information correlates with the `Non-`

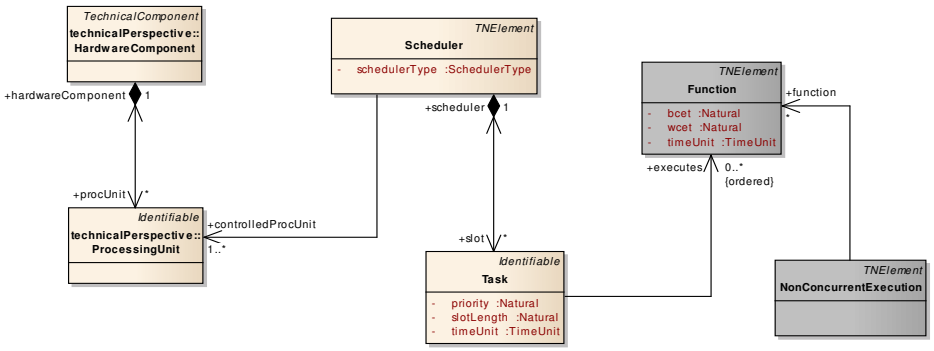


Figure 5: Excerpt of the metamodel extension showing `NonConcurrentExecution` relation

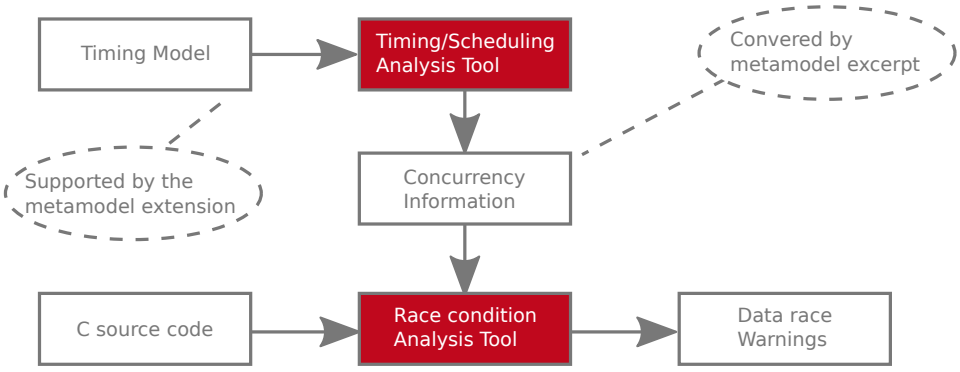


Figure 6: Conceptual tool chain

`ConcurrentExecution` relation between `Functions` as depicted in the metamodel excerpt in Figure 5. The race condition analysis tool (for example the Bauhaus toolsuite) picks up this data, excluding some potential data races in its analysis. Note that in order to ease exposition, we simplified the class diagram. Actually, besides the functions and concurrency relation, also paths to the C source code files and the signatures of the methods implementing the functions, are handed over to the race condition analysis tool. This is necessary in order to map the concurrency information to the source code implementation level.

Finally, we like to point out that though we stressed SymTA/S and Bauhaus as examples supporting the use case, the tool chain shown in Figure 6 can also be setup with different tool candidates. The definition of the tool chain on a conceptual level based on the vocabulary defined by the metamodel, allows us to assess whether another tool can fill in either of the two roles.

For example the timing analysis based on model checking proposed in [SRGB13], is a

viable alternative. In contrast to potential over-approximations in analyses based on analytical methods like SymTA/S, this approach delivers exact results. This is due to the abstraction of analytical approaches because of the inevitable limited accuracy of the underlying formalisms to capture complex dependencies between different events. On the other hand analysis approaches based on model-checking are able to capture those event dependencies while constructing the state-space of the system. Consequently, more potential data races can be excluded. However, there is no free lunch as model-checking suffers from potential state-space explosion.

With regard to race condition analysis, the tool Gropius from the University of Kiel can be used as a replacement. These tools have their own strengths and weaknesses. For example a trade-off exists between analysis time and minimizing the number of false positives with regard to detected data races.

5 Conclusion

We presented a strategy for creating tool chains for the design and analysis of multi-core systems. Thanks to a layered approach, where tool chains are first designed on a conceptual level backed by a metamodel, a chain is independent from a particular tool integration technology. This allows us to choose a technology that fits best for each pair of tools in a tool chain. Focusing on the conceptual layer, we presented parts of the metamodel. Further, we presented a particular use case for tool integration from the ARAMiS project, where a race condition analysis tool makes use of the analysis results of a timing and scheduling analysis tool. Based on this use case, we classified the information exchange in this tool chain according to the concepts of the metamodel.

Currently, we are working on the transition to the implementation level. The tools SymTA/S and Bauhaus are used as candidates for this manifestation of the tool chain. A text file with a simple format contains the exchanged information in this prototype implementation.

References

[ATE08] The ATESST Consortium. *EAST ADL 2.0 Specification*, February 2008.

[AUT09] AUTOSAR GbR. *Software Component Template*, November 2009. Version 4.0.0.

[BDH+12] Manfred Broy, Werner Damm, Stefan Henkler, Klaus Pohl, Andreas Vogelsang, and Thorsten Weyer. Introduction to the SPES Modeling Framework. In Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy, editors, *Model-Based Engineering of Embedded Systems*, pages 31–49. Springer, 2012.

[DHH+11] Werner Damm, Hardi Hungar, Stefan Henkler, Thomas Peikenkamp, Ingo Stierand, Bernhard Josko, Philipp Reinkemeier, Andreas Baumgart, Matthias Büker, Tayfun Gezgin, Günter Ehmen, Raphael Weber, Markus Oertel, and Eckard Böde. Architecture Modeling. Technical report, OFFIS, March 2011.

[Gro07] The SPICE User Group. Automotive SPICE Process Assessment Model. Technical report, 2007.

[Int] International Organization for Standardization (ISO). *ISO 26262: Road vehicles – Functional Safety*.

[JS11] Dave Johnson and Steve Speicher. Open Services for Lifecycle Collaboration Core Specification Version 2.0. Technical report, 2011.

[NM90] Robert H.B. Netzer and Barton P. Miller. On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions. In *In Proceedings of the 1990 International Conference on Parallel Processing*, pages 93–97, 1990.

[Obj08] Object Management Group. *OMG Systems Modeling Language (OMG SysML ™)*, november 2008. Version 1.1.

[Par09] Project SPEEDS: WP.2.1 Partners. D.2.1.5 SPEEDS L-1 Meta-Model. Technical report, The SPEEDS consortium, 2009.

[Ric05] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, Germany, 2005.

[SRGB13] Ingo Stierand, Philipp Reinkemeier, Tayfun Gezgin, and Purandar Bhaduri. Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, 2013.