Handling Complex Dependencies in System Design

Mischa Moestl, Rolf Ernst
Institut für Datentechnik und Kommunikationsnetze
Technische Universität Braunschweig
Braunschweig, Germany
{moestl | ernst}@ida.ing.tu-bs.de

***Abstract*—In this paper we describe a novel strategy to reveal and handle complex dependencies in an incremental and distributed design processes even under the ubiquitous presence of uncertainties concerning model and design. We demonstrate in a case study how to handle epistemic design uncertainty in an iterative process and present how it is possible to selectively exclude dependency paths under certain concerns such as timing by including third party analysis results based on the used models into the dependency analysis. Since the implementation of our approach relies on modern graph analysis libraries it can scale to realistic problem instances.**

I. Introduction

The behavior of complex systems such as cars, planes, manufacturing machinery, etc. is nowadays more and more entirely software dominated. Further the established design processes are characterized by abstraction, both in model and in design. Additionally it also exhibits ubiquitous design uncertainty during system development due to not yet made design decisions or lack of knowledge about them in different company divisions. Especially tentative or unsettled design decisions introduce risk into complex distributed designs and can thus jeopardize a project even in very late design and verification steps [1]. This makes designs prone to various types of hidden dependencies, especially layered system designs with distributed applications and RTEs such as AUTOSAR. In consequence there is a need to account for these epistemic uncertainties in the system model, yet note that probabilistic uncertainties, e.g. hardware faults and their effects are an orthogonal problem.

For instance in automotive architectures signal flows from sensors to actuators traverse several Software Components (SWCs), the Run-Time Environment (RTE), CPUs as well as bus and network layers of a vehicle. Here e.g. the end-to-end timing behavior of a signal flow is often abstracted away entirely. The organizational structure of development teams often contributes to such effects, since model data is often distributed within an organization yielding uncertainties about whether a dependency to other artifacts in the system exists. In consequence abstraction and hierarchy between the layers create caveats in the design by introducing hidden dependencies. This can especially undermine properties like safety, dependability or timing requirements of sub-systems.

Therefore a methodology for design and analysis techniques must address the issues of handling multiple abstraction levels of complex systems while grasping design uncertainty in an incremental design process. The methodology must further be able to iteratively bring more fine grained information into the model where necessary, i.e. when design decisions are fixed and are no longer uncertain or if additional knowledge is necessary for analysis. It is then able to resemble a typical development flow from few definite decisions in the beginning to a fully specified system.

This work was supported by the DFG Research Unit Controlling Concurrent Change (CCC), funding number FOR 1800.

Contribution: We propose a design and analysis technique to address these core issues. Our methodology directly models the epistemic design uncertainty and is able to iteratively bring more fine grained information into the model where necessary. We achieve this while maintaining compatibility to conventional design models applied in systems design.

The impact on the design process and model are twofold: First, the methodology allows designers to detect and pick-up consequences a possible mapping might have, e.g. w.r.t. safety concerns. Second and most notably, picking up possible dependencies in an early design phase allows either to circumvent them by choosing an allocation of system elements, such that a dependency cannot manifest or that safeguards can be brought into the design to guard against malicious influence of a dependency. Iteratively adding information to our system model and the emerging description of the system allows a subsequent reevaluation of the system until all possibly hazardous dependencies are handled. Our approach relies on graph theory to express design uncertainty, yet leverages the knowledge that is already contained in domain-specific models.

II. Related Work

In the context of safety-critical systems methods like Fault-Tree Analysis (FTA) [2] or Failure Modes and Effects Analysis (FMEA) [3] can be used during the development process to handle dependencies. Yet they suffer from insufficient knowledge in early stages of the design and are difficult to maintain if a subsystem is reused in a different design since the dependencies of the fault-tree are hard to maintain. This is particular true when different aspects of the system are described with decoupled models in distributed design teams.

Other approaches to handle dependencies in real-time systems such as presented in [4] are based on the failure semantics for real-time systems. Yet the approach is software centric and neglects cross-layer issues, e.g. is a software engineer unaware of the fault behavior of a certain platform such that certain failure semantic can be observed on the software layer. However we see our proposed method as a probable way to obtain these relation by investigating the actual dependencies between the functionalities described by software and hardware platform.

III. Handling Complex Dependencies

Direct dependencies between elements of a system are usually handled by constructing a model that captures certain aspects at a chosen abstraction level. However, due to the abstracting nature of these models it is impossible to capture all effects in one model. It is thus necessary to combine knowledge that can be gained from the already established aspect specific models present in common modeling methodologies, e.g. EAST-ADL [5], AUTOSAR [6] or AADL [7].

A. Leveraging knowledge from different models

The advantage that we exploit here is that these models can be represented as digraphs with directed edges, where e.g. nodes represent functions or features and edges their direct dependencies between each other. This works for high-level models such as function-feature networks (e.g. [7], [5], [8]) but also for network topologies, task graphs and platform models (e.g. [9] [10]).

However, cross-layer dependencies regarding timing, reliability or safety go beyond the effects captured in individual models. In order to bring the knowledge of multiple architecture-model layers together, we merge their digraph representations by introducing a mapping between them.In that sense a mapping can be understood as "is implemented on", e.g. a system function is decomposed and implemented on a network of logical functions which again are mapped onto an RTE architecture that constitutes the execution environment of the logical functions. Connecting individual architectural-model layers via mappings provides the possibility to reveal cross-layer dependencies by spanning the transitive closure over intra-layer and mapping dependencies. To capture this formally, we define:

Definition 1. A *system architecture* S is a digraph (V, A) s.t.
1) the set of nodes $V = \{V_i\}_{i=1}^n$ where $n \in \mathbb{N}^+$ and each V_i represents the node set of an architecture model-layer;
2) the set of arcs A is partitioned into $n{+}m$ subsets, namely for each V_i exactly one subset A_i relating nodes in V_i according to the semantics of the model; and m subsets relating the nodes V_i and V_j (for each model j model i maps to).
For all $1 \leq i, j \leq n$, (V_i, A_i) defines a digraph G_i called an *architectural model* of S. The subset of A relating V_i and V_j is called *mapping of model i onto model j*.

Furthermore we can define a dependency within S as follows:

Definition 2. A *dependency* between two elements $v_x, v_y \in V$ in a system architecture S is a path p leading via an alternating sequence of vertices and arcs: $v_0, a_0, v_1, \ldots, a_k, v_k$ (for $k \in \mathbb{N}^+$) leading from v_x to v_y.

Note that the existence of a dependency is a purely syntactic criterion that only expresses the fact that there is a potential influence between v_x and v_y. Although we use the term *layer* and enumerate them, we do not impose a strict hierarchy of individual architecture-model layers or a fixed set of models since these can be domain specific. In fact the topology of mappings between multiple layers can be arbitrary. It is even possible to have fork-join constructs of mapping relationships or feedback loops over certain models, as depicted in Fig. 1. For instance the later is of interest, when a system manipulates its physical environment via actors and "reads" this information back thus influencing the functional behavior of the system, while the former is reasonable if memory and processing is captured in separate models instead of a common RTE model.

Although this allows to reveal complex dependencies that have a cross-layer nature, it does not allow to model design uncertainty yet.

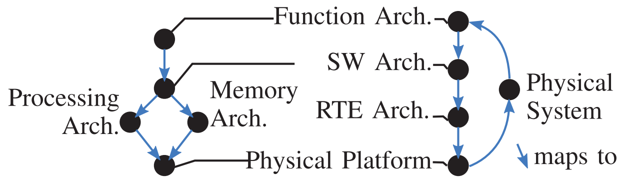

Fig. 1. Different system model compositions due to different mapping relationships between architectural models

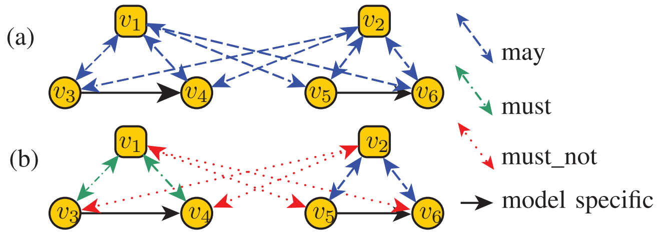

Fig. 2. Mapping after adding design knowledge, reducing uncertainty since element v_1 now only depends on v_3 and v_4

B. Handling Uncertainty - Different Mapping Types

Mapping reflects the fact that some models represent a specialization or refinement of another one regarding certain aspects of the design. This is a common practice that is for example applied in the model stack of [5] for the automotive domain. In order to account for design uncertainty in this process without introducing optimistic assumptions we categorize mapping relations in three types.

First of all the intuitive type that represents a fixed design decision, meaning that a specific artifact from one model is definitely implemented on one or more specific artifacts of the other model, i.e. the mapping must occur. Consequently we refer to this mappings as $must$ mappings. Early settled $must$ mappings include, e.g. that software drivers for peripherals have to be placed on the hardware components that actually feature them.

On the contrary if it is known that a specific mapping can and will not occur, i.e. it can be ruled out on the basis of design decisions that the two artifacts are in relation, we classify the mapping as $must\_not$. For instance if a SWC is not available for a certain CPU type.

All remaining mappings where no definite answers are available stay in a state that also reflects this. We refer to this type of mapping as may, since the mapping might appear due to decisions further on in the design process or it might be even dynamic at runtime. Formally we grasp them as:

Definition 3. A *mapping* forms a complete bipartite subgraph $M^i_j = (V_i, V_j, A^i_j)$ in S consisting of the two node sets V_i and V_j where each arc pair $(v_i, v_j), (v_j, v_i)$ in A^i_j is of one of the three mapping types $may, must, must\_not$.

$$A^{i,j} = A^{i,j}_{may} \cup A^{i,j}_{must} \cup A^{i,j}_{must\_not} \qquad (1)$$

Upon defining a mapping between two models, all nodes from the source model are mapped to all nodes from the destination layer with mappings of the type may. Simply put we always assume dependence through may mappings were independence cannot be proved straight away. This goes along the lines of common safety standards where dependence has to be assumed as long as independence cannot be proved, i.e. our conservative assumption is backed by this fact ([11], [12]).

In the course of refining a design, it is then possible to iteratively and selectively add more information to eliminate uncertainty, i.e. mapping types can be changed from may to $must$ or $must\_not$ based on design decisions or additional knowledge, as depicted in Fig. 2 (b).

The advantage of this approach is that it enables our analysis methodology to be conservative in nature, since it guarantees that adding more knowledge to the model will only lead to a lower number of certainly dependent elements (cf. Fig. 2).

The same holds, if certain structures of a model are refined and modeled in more detail, e.g. to enable an independence proof for certain elements sharing the respective one. This is achieved by annulling all mappings to the old element and replacing it with a new may-mapped subgraph refining the model. However, it has to be ensured that the new subgraph structure specializing a node must preserve all intra layer in and output relations. In consequence all incident mappings are reinitialized as may from each node in the incident layer, ensuring that all possible dependency paths can be captured.

IV. Cross-Layer Dependency Analysis

In order to handle dependencies during a development process, e.g. for safety purposes, it seems promising to maintain a list of all possible dependency paths to avoid manifesting them. Especially w.r.t. safety validation of a design where freedom from interference is the ultimate design goal [11]. In that context every dependency must be treated as potential interference. Although tempting, enumerating all paths between two nodes in S suffers from combinatorial explosion.

Even if cyclic paths are excluded, a fully connected graph with only $|V| = 10$ nodes has a total of $|V|^2$ bidirectional arcs, resulting in $|V|^2! \approx 9 * 10^{157}$ simple paths in S.

Instead we explore the system model S by breadth-first search (BFS) or depth-first search (DFS) which feature a reasonable run-time and memory complexity. The result is a boolean *map* that specifies whether a node v_j is reachable from v_i. These maps can be stored and maintained efficiently at a minimum overhead. As a byproduct of constructing this map it is further possible to obtain the shortest dependency path to any dependent element. It goes without saying that for our purpose we only investigate cycle-free paths within S, as otherwise infinite loops could occur.

In order to maintain a structured view on already existing dependencies due to fixed decisions and possible dependencies modeled by may mappings during a development process, different views on the system graph S are possible. This is achieved by only considering the respective mapping arcs. Such sub-graph filtering can be efficiently applied without the necessity to copy the actual data of S in the implementation. This allows to construct three primal views on the dependencies of a system S. First of all it is possible to consider the consequences of all already made design decisions by obtaining the BFS-tree only considering the $must$ mapping, excluding the may and $must\_not$ mappings. Second, also the map of all system elements a certain element possibly depends on is available by only considering may arcs in the BFS. As a third option also a combined strategy of the two former is possible. Therefore only the arcs in the $must\_not$ sets are omitted when traversing S via BFS.The latter strategy then reveals all reachable and possibly reachable elements a certain element depends on. Already this pure syntactic evaluation of the system model allows to identify connected components of a system that must fulfill the same requirements as they inherit it from models that map on them. For instance if a safety-critical function is placed somewhere in a design it can be quickly assessed where the functional safety requirements would propagate in the whole system, or where isolation between different Safety Integrity Levels (SILs) would be necessary. Since we do not enumerate all paths explicitly but only the reachable set we follow an incremental strategy here where always the shortest dependency path is investigated first, since it is ab byproduct of computing the reached set anyway and comes with no additional computational costs.

V. Effect Specific Dependencies

Studies of system-function architectures of vehicular platforms indicate that within the function architecture already up to 85% of the vehicle functions depend on each other ([13], [8]). This is further fostered by highly networked platforms but also feedback through the physical environment implying a system graph in which almost everything depends on everything else, at least indirectly.

A. Including Semantic Information

So far we only syntactically interpreted the model neglecting the semantic properties within the individual architecture models and mapping relations. Based on the semantic properties of the employed models it is possible to bound the dependence between model elements by concern specific analysis and thus also bound the exerted interference. Hence an exploration of the model is necessary to resolve whether a dependency is an actual source of interference or whether the dependency path can be neglected for a particular concern. Since this rests on specific concerns, e.g. timing or hardware faults, triggering a concern specific analysis is then necessary to unravel such paths. In order to reason about a design in total, the system model must be annotated with the analysis results to be able to exclude annotated arcs in the next iteration.

We consider the small case-study system from Fig. 3 here, where the independence between the functions f_1 and f_3 should be examined. Performing the syntactic analysis run reveals that both functions are in each others reachable set although functionally independent. We assume f_3 to be time critical and neglect further concerns for simplicity. Thus no dependence or interference regarding timing must exist. Since RTE_2 is in the set of reachable nodes via $must$ mappings for both investigated functions and the shortest path traverses it, we analyze the scheduling behavior whether an actual dependency between τ_1 and τ_3 exists. As the scheduler in this example is a property of the RTE, the timing concern is a property dealt with on the software and RTE-architecture model. However, which additional information can be leveraged in general to achieve (semantic) independence proofs regarding syntactic dependencies is a property of the employed models. For the timing model applied here, we resort to the task level for the definition of timing non interference.

Definition 4 (Timing Non-Interference)**.** Let τ_i and τ_j be two tasks mapped to the same processing resource. Then τ_i does not interfere with task τ_j if two different activation traces $\sigma_i \neq \sigma_i^{'}$ do not yield a different termination trace $\omega_j \neq \omega_j^{'}$ of τ_j.

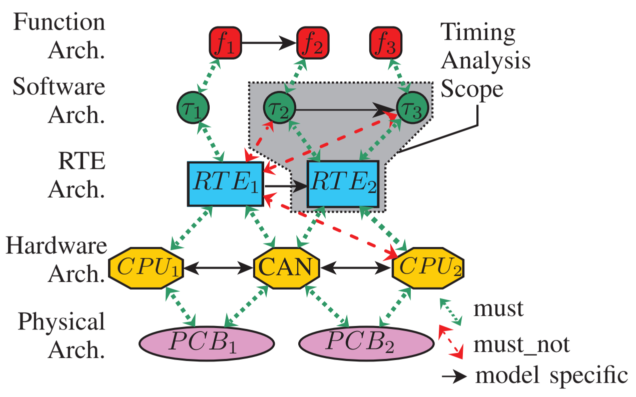

Fig. 3. Model of the example system (*may* mappings omitted for clarity)

Definition 5 (Activation and Termination Trace). An *activation trace* σ_i and a *termination trace* ω_i of a task τ_i are functions $\sigma_i : \mathbb{N} \rightarrow \mathbb{R}^+ \times \mathbb{R}^+$ and $\omega_i : \mathbb{N} \rightarrow \mathbb{R}^+$ where $\sigma_i(n) = (t, C)$ denotes that the n-th event occurring at time t requires a processing time C and $\omega_i(n) = t'$ the time the n-th activation of τ_i is completed.

The property of Definition 4 can e.g. be fulfilled by a static-priority scheduler, as common in AUTOSAR. In the example in Fig. 3 this is the case if τ_3 has a higher priority than τ_2 on RTE_2 and cannot experience blocking from τ_2.

B. Including Effect Analysis Results

This information can now be added to the system model and stored on the node representing RTE_2. For a new BFS run to investigate timing independence, this information can now be leveraged by not traversing the backward mapping arc $(RTE_2, \tau_3) \in A^{i,j}$ whereas i and j represents the RTE and task layer respectively. This then eliminates all timing dependency paths containing the subpath $p = \tau_2, (\tau_2, RTE_2), RTE_2, (RTE_2, \tau_3), \tau_3$ and the remaining paths or practically the next shortest path can be considered. As a side effect the system model can automatically provide the set of elements such an analysis results depends on if the model underlying the analysis is directly contained in S. In the timing analysis example here this is the merged set of all reachable nodes originating from τ_2, τ_3 and RTE_2.

VI. Scalability Case Study

In order to demonstrate the scalability of our approach implemented on top of a standard graph library described in [14] we measure the peak amount of RAM consumption to hold a model and compute the reachable set for a randomly picked node. For this experiment we generated synthetic models that resemble a full automotive platform. For simplicity we assume only three architecture-model layers here and chose the parameters for generating arcs within one model layer such that they resemble realistic systems (e.g. [8]). The number and the size of the resulting graphs are listed in Table I.

On a Core-i5 system running Linux an average of 3 GiB is necessary to host the model and perform a BFS to compute the reachable set of a randomly picked node which can be seen reasonable given the size of the problem.

A possible memory and runtime improvement is to assume one mapping type implicitly and not storing the respective arcs explicitly, since this would significantly improve the

TABLE I
Parameters for randomly generate systems

	Nodes	Arcs
Vehicle Functions	100	800
Mapping	-	64.000.000
Runnables & Signals	640.000	1.920.000
Mapping	-	76.800.000
ECUs & Buses	120	360
TOTAL	**640.220**	**142.721.160**

sparseness of the graph. Which type however strongly depends on whether from scratch designs dominate with a enormous amount of *may* mappings or iterative designs where the majority of mappings is already settled.

VII. Conclusion

We presented a system modeling methodology that is able to handle dependencies in complex systems efficiently, while also being able to automatically deduce all elements of a system another element depends on in a conservative fashion. The presented analysis capabilities have their foundation on the conservative assumption, that independence has to be explicitly proved, either by eliminating uncertainty or through semantic analysis. In either case the method relies on the fact that including more model information only makes the result better, in the sense that more independence can be argued with the additional information, never the opposite. This fact supplements current development processes, where a lot of decisions are only iteratively made and sometimes no overview of dependencies in the overall system exist since the information is spread over various branches of a company or different suppliers. Here linking the domain specif models by the mapping approach and exploiting the resulting system model can lead to a significant increase in design knowledge, productivity and safety of the resulting system.

References

[1] S. Arora, P. Sampath, and S. Ramesh, “Resolving uncertainty in automotive feature interactions,” in *Requirements Engineering Conference (RE)*, Sep. 2012.
[2] IEC, “Fault tree analysis (FTA),” The International Electrotechnical Commission, Tech. Rep. IEC 61025, December 2006.
[3] ——, “Analysis techniques for system reliability - procedure for failure mode and effects analysis (FMEA),” The International Electrotechnical Commission, Tech. Rep. IEC 60812, January 2006.
[4] H. Ding and L. Sha, “Dependency algebra: a tool for designing robust real-time systems,” in *RTSS 2005*, Dec. 2005.
[5] *Specification EAST-ADL V2.1.12*, EAST-ADL Association, Nov. 2013.
[6] *AUTOSAR Specification Release 4.2*, AUTOSAR, 2015.
[7] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.
[8] A. Vogelsang, S. Teuchert, and J.-F. Girard, “Extent and Characteristics of Dependencies Between Vehicle Functions in Automotive Software Systems,” in *MiSE*, 2012.
[9] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, “System level performance analysis - the SymTA/S approach,” in *IEE Proceedings Computers and Digital Techniques*, 2005.
[10] P. Huang, P. Kumar, N. Stoimenov, and L. Thiele, “Interference constraint graph a new specification for mixed-criticality systems,” in *(ETFA 2013)*, Sep. 2013.
[11] *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2nd ed., IEC, April 2010.
[12] *ISO 26262 - Road vehicles – Functional safety*, International Organization for Standardization - ISO, 2011.
[13] A. Vogelsang and S. Fuhrmann, “Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study,” in *Requirements Engineering Conference (RE)*, Jul. 2013.
[14] B. Dezső, A. Jüttner, and P. Kovács, “LEMON – an Open Source C++ Graph Template Library,” *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 5, Jul. 2011.