SICS Software-Intensive Cyber-Physical Systems (2021) 35:185–199
https://doi.org/10.1007/s00450-021-00428-2

REGULAR PAPER

Automating integration under emergent constraints for embedded systems

Johannes Schlatow[1] · **Edgard Schmidt**[1] · **Rolf Ernst**[1]

Received: 3 August 2020 / Accepted: 20 September 2021 / Published online: 23 October 2021

Abstract
As embedded applications are subject to non-functional requirements (latency, safety, reliability, etc.) they require special care when it comes to providing assurances. Traditionally, these systems are quite static in their software and hardware composition. However, there is an increasing interest in enabling adaptivity and autonomy in embedded systems that cannot be satisfied with preprogrammed adaptations any more. Instead, it requires automated software composition in conjunction with model-based analyses that must adhere to requirements and constraints from various viewpoints. A major challenge in this matter is that embedded systems are subject to emergent constraints which are affected by inter-dependent properties resulting from the software composition and platform configuration. As these properties typically require an in-depth evaluation by complex analyses, a holistic formulation of parameters and their constraints is not applicable. We present a compositional framework for model-based integration of component-based embedded systems. The framework provides a structured approach to perform operations on a cross-layer model for model enrichment, synthesis and analysis. It thereby provides the overarching mechanisms to combine existing models, analyses and reasoning. Furthermore, it automates integration decisions and enables an iterative exploration of feasible system compositions. We demonstrate the applicability of this framework on a case study of a stereo-vision robot that uses a component-based operating system.

Keywords Model-based integration · Component-based systems · Distributed real-time systems · Backtracking

1 Introduction

Autonomy is a current trend in Cyber-Pysical Systems (CPS) as illustrated by the numerous automated and autonomous driving initiatives. Here, autonomy involves decision-making in reaction to (unforeseen) events in the system function, the platform and the environment, which requires platform adaptivity to deal with these dynamics in order to maintain a safe operation [19].

As CPS are subject to non-functional requirements such as latency, safety and reliability, the correct operation of such a system also relies on emergent and platform-specific properties. The state of the art is that these requirements are handled by rigorous development processes that involve the design, integration and testing of the system and its subsystems. By integration and testing, we refer to the composition, distribution and parametrisation of a system and its assurance w.r.t. application and system-level requirements. These processes also involve (manual) decisions, predictions and abstractions based on expertise, which may lead to iterations if it turns out that the integrated system cannot satisfy all requirements in the end. Tracking down what decisions or parameters need to be tuned to solve integration issues is often a manual debugging procedure. With increasing adaptivity and platform complexity, however, predicting all possible adaptations (i.e. changes to the system at runtime) will not be an option any more. A system must therefore be able to deal autonomously with functional adaptations, unforeseen platform modifications and change of available implementations. In the general case, this necessitates repeating the integration procedure to

This work was supported by German Research Foundation (FOR 1800).

✉ Johannes Schlatow
johannes@schlatow.name

Edgard Schmidt
schmidt@edik.ch

Rolf Ernst
ernst@ida.ing.tu-bs.de

[1] Institute of Computer and Network Engineering, Technische Universität Braunschweig, Brunswick, Germany

provide proper assurances. If this integration procedure could be automatically performed in the field, i.e. on the CPS itself or with assistance of cloud or edge-computing, we would gain a new kind of platform autonomy for CPS.

This in-field integration was the central research question of the DFG research unit *Controlling Concurrent Change (CCC)* that we will explain more detailed in Sect. 2. In order to achieve this, we thus need to automate integration activities such that assurances can be provided before the system is reconfigured. Model-based analyses are able to provide assurances if a model (view) with the respective semantics is available for the system. Examples for this are fault-tree analyses (FTAs), failure mode and effects analyses (FMEAs) or response-time analyses (RTAs).

In the scope of this paper, we focus on RTAs for assurance of worst-case latency requirements as a representative example where formal models and automated analyses are readily available. The challenge in this regard is to acquire all the required views and couple them to the integration decisions and adaptations that cause side effects on other views. Normally, a holistic model incorporating all relevant views is formulated to approximate and predict these effects and enable global optimisation. Such an approach, however, will be specifically tailored to a particular application domain and platform such that semantic extensions become rather expensive or abstract. In practice, one will often find related work that address sub-problems in very detail but which cannot be trivially extended to cover the full spectrum of the own problem.

The main question that we are addressing in this paper is thus how we can combine existing models and methods into an overarching framework for automating the integration procedure while being able to provide assurances for critical requirements. The focus in this regard lies on feasibility rather than optimality. We contribute to this question by introducing a cross-layer model on which existing methods can be applied via a formalised set of operations. Furthermore, a depth-first search algorithm is defined which keeps track of the performed operations in order to selectively roll-back operations that need to be revised. Together, this builds the foundation for an open-source software framework that automates the integration procedure of CPS for finding an initial deployment from scratch and for performing incremental adaptations for an existing deployment. We demonstrate the application of this framework on a realistic case study from the CCC project.

The remainder of this paper is structured as follows: We describe the general architectural approach supporting in-field changes in Sect. 2 before we introduce an illustrative use case in Sect. 3 that we use as a running example. In Sect. 4, we review the related work and present our framework in Sect. 5. We apply our framework to the presented use case in Sect. 6 and draw our conclusions in Sect. 7.

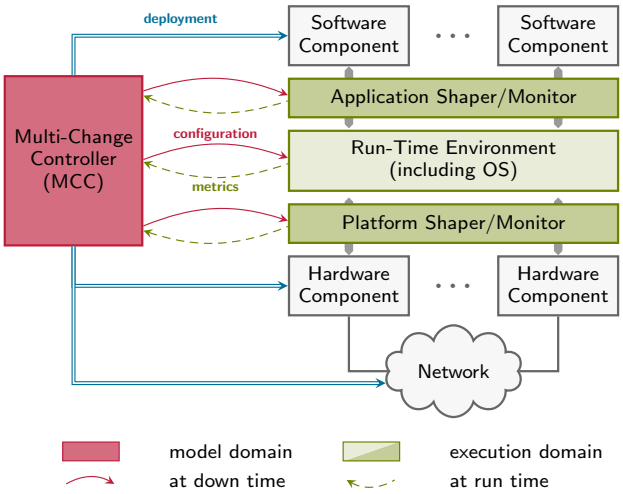

Fig. 1 Architectural approach of CCC with model domain (red), execution domain (green), and changing software/hardware components (gray). [18] (colour figure online)

2 General approach

In the domain of CPS, we commonly deal with statically configured systems that, once deployed, only support minor adaptations at runtime. As mentioned in the previous section, significant changes to a CPS require redoing an integration procedure that provides the necessary assurance for critical requirements such as worst-case latency. Instead of (manually) performing the integration for every change and target platform in the lab, we want to equip CPSs with integration capabilities to autonomously manage any changes. Our architectural approach for supporting this is depicted in Fig. 1. The system is split into a model domain and an execution domain. The runtime environment (RTE) including the Operating System (OS) resides in the execution domain. It runs on a set of hardware components that are connected via one or multiple networks (representing the processing platform) and hosts a set of software components (representing the applications). The RTE is optionally augmented by shaping and monitoring mechanisms that enforce and observe runtime behaviour. The model domain comprises the Multi-Change Controller (MCC), which is a software module that operates either as a background process on the same processing platform or in the edge/cloud. The MCC takes full control over the deployment of software components and the system configuration by handling all change requests to the system. Change requests can be issued internally by the execution domain or by external events (e.g. user interaction). It therefore performs automated integration to find a feasible solutions to the change request and model-based analyses to provide assurance for critical requirements. Requests can be rejected, if no acceptable solution is found.

In the DFG research unit CCC, two application domains have been explored: automated driving and space robots. When it comes to automated driving, we are facing complex and time-sensitive applications in a safety-critical environment. In this setting, a modularisation of Advanced Driver Assistance Systems (ADAS) together with the in-field integration approach could enable user-determined functionality (customisation) [18]. For space robots, we are more concerned about heterogeneous resource-constrained devices in a mission-critical setting where solutions to unexpected situations (e.g. hardware failures, environmental conditions, power shortage) shall be found autonomously [8].

3 Illustrative use case: space robot

In this section, we present a use case on which we base the presentation of our work. For this purpose, we summarise the necessary information of the underlying RTE, introduce the application scenario, and provide a formal problem statement.

3.1 Runtime environment

Building safe and secure systems that dynamically change is a very challenging and multidisciplinary objective. As policy and behaviour are not fixed and verified at compile time any more, assumptions made by model-based approaches must be maintained at runtime. Microkernel-based architectures are a popular approach for maintaining strong process isolation and strict access control by OS design. While component-based design is mainly a modelling concept, there are also component-based approaches to OS construction [15,27] that enforce the component concept at runtime. In such systems, components represent atomic building blocks (compiled design artefacts) that can be plugged together at runtime by connecting their Inter-Process Communication (IPC) interfaces. We use the *Genode OS Framework* [10], which follows the microkernel approach. Genode is a component-based OS in which each component operates in a separate address space. Components interact and communicate with each other via service interfaces of different types. A service interface type specifies the protocol of the interacting components on a syntactical level, i.e. what signals and data types are exchanged. A service provided by a component can be used by one or multiple client components. Following the principle of least privilege, a component is explicitly granted the required connections to the services of particular components. This component-based system is hierarchically structured in order to minimise the Trusted Computing Base (TCB) of each application [13] and therefore designed for security. In contrast to a layered architecture there is no single entity (e.g. middleware) that is authorised for the delegation of resources and services. This also means that software components cannot be easily mapped or migrated to other processing nodes without modifying the system composition. Instead, distribution is achieved by inserting *proxy components* that translate service interfaces (using IPC) into network communication and vice versa [11]. As run-time dependencies are therefore explicitly granted, there is no uncertainty in the system composition and the possible component interactions.

Components can be further categorised into application components, device drivers, resource multiplexers, and protocol stacks. While *application components* have an application-specific purpose, components of the latter three categories can be universally used as library components. A *device driver* accesses a particular hardware device (e.g. network interface controller) and provides a service interface to another component. Many device drivers only allow a single client component so that a *resource multiplexer* is required to coordinate the access from multiple clients. A *protocol stack* translates a service interface into an interface of a different type.

By instantiating and interconnecting different components, a particular functionality is implemented on the system. The system composition is specified in XML to allow (re)configuration of the system at runtime. Note, that most components (apart from device drivers) can be instantiated (i.e. replicated) multiple times and therefore used in different contexts. For time-sensitive applications, scheduling priorities can be assigned to components and their processor affinity can be set.

3.2 Application scenario

Our application scenario originates from the CCC project in which a space robot was developed for exploring an unknown environment in search for certain objects (cf. [8]). The robot is equipped with a stereo camera and executes various image-processing algorithms. It is further designed as a processing platform that can adapt to function changes, platform changes and changes in the environment conditions. This platform consists of two nodes that are interconnected via Ethernet as illustrated in Fig. 2. It is therefore representative for modular and heterogeneous embedded systems that integrate multiple off-the-shelf processing nodes into their processing platform. In this use case, each processing node has a static subsystem and a dynamic subsystem. The dynamic subsystem shall be controlled by our framework whereas the static subsystem hosts very basic functionality (e.g. the Ethernet driver). Node 1 is further divided into two symmetrical processing cores and an optional Field Programmable Gate Array (FPGA) that acts as a reconfigurable co-processor.

In this paper, we restrict our scope to two different functions of the robot: An object detection algorithm for

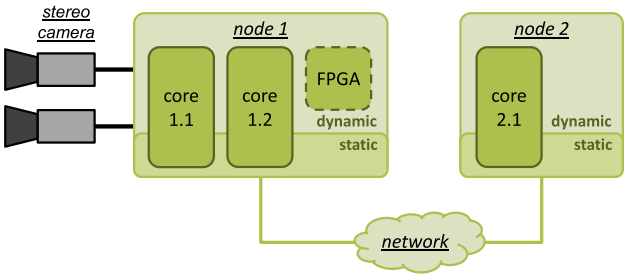

Fig. 2 Heterogeneous computing platform of the stereo-vision system

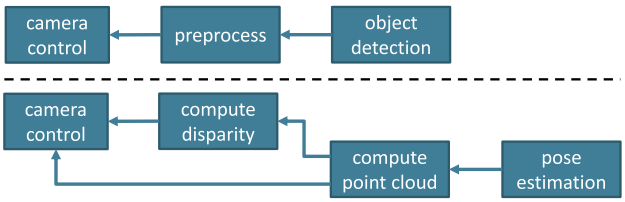

Fig. 3 The platform executes two different functions: object detection (top) and pose estimation (bottom)

identifying objects in the environment and a pose estimation algorithm for picking up objects. Figure 3 depicts the high-level architecture of both functions as interconnected function blocks and their dependencies. The object detection requires some preprocessing of the camera images as well as control access to the camera. The pose estimation algorithm is based on a point cloud describing the prospective objects. In order to compute the point cloud, a raw camera image as well as a disparity image is required that contains the distance of each pixel. In both cases, this results in a processing chain with a latency requirement: The object detection must be performed with at least 10 frames per second (fps) to allow a minimum speed of the robot. On the other hand, the pose estimation shall be executed with at least 5 fps. We thus define the latency requirements of 100 ms and 200 ms respectively.

There is a variety of implementation options for the particular function blocks. For instance, preprocessing could be implemented in software or with assistance of the FPGA. There can also be different algorithms that trade processing time with, e.g., robustness. As an example, there are numerous stereo matching algorithms [12] that optimise and refine the disparity at the cost of computation time and power consumption. An adaptive platform should be able to choose from different implementations dependent on the conditions, available resources, and performance constraints. Alternative implementations, however, may require different resources, libraries, and privileges. Instead of precomputing possible variants in advance, we want to automate the integration process and thereby handle unexpected changes. Such changes can be triggered from monitoring infrastructure that observes platform and environment parameters at runtime. For our investigations, we consider changes in function, platform and environment: We address function changes by switching between object detection and pose estimation. By toggling the availability of the FPGA (e.g. caused by power restrictions or hardware failures), we accommodate for platform changes. Environment changes are incorporated by sensing whether the robot enters an area with high particle flux (e.g. caused by a solar flare) in which case it must choose a more reliable implementation. The goal of the integration process is to find and select a suitable implementation and a distribution such that the intended function can be executed on the given platform and latency requirements are satisfied. In order to provide assurance even if the implementation deviates from the model, runtime monitoring should be applied to observe critical model parameters. For instance, execution times can be monitored to maintain the assurance of latency requirements [25].

3.3 Problem statement

Based on the use case presented above, let us briefly formulate a more concrete problem statement. We understand the integration process as an incremental enrichment and refinement of architectural models which starts with a platform- and implementation-independent functional view. Other views are derived during this process: An *implementation view* is acquired by distributing function blocks on the target platform and by selecting available implementations. In our use case, this view serves as a basis for reconfiguring a component-based OS. For assurance of latency requirements, we further need to derive a *timing view* that provides an in-depth look at the execution behaviour of the implementation.

The inputs are given as a predetermined *function architecture* according to Definition 1 of the system (cf. Fig. 3), a *platform model* according to Definition 2 that specifies the structure of the available processing resources (cf. Fig. 2), and latency requirements that specify the maximum allowable latency for a chain of function blocks.

Definition 1 A *function architecture* is a directed graph $G_{func} = (V, A)$ in which the nodes $v \in V$ represent function blocks and directed edges $a \in A$ represent function dependencies. An arc $a = (u, v)$ denotes that u depends on a function provided by v.

Definition 2 A *platform model* is provided as a bipartite graph $G_{platform} = (U, V, E)$ where U denotes the set of processing nodes, V the set of communication resources (networks, buses), and E determines what communication resources are accessible by what processing node. A processing node $u \in U$ is characterised by a tuple (F_p, c, R_p) in which F_p is a set of provided hardware capabilities (flags) that abstract the microarchitecture (e.g. instruction set, available peripherals, CPU frequency), c is the number

of symmetrical processing cores, and R_p specifies the amount of provided resources (e.g. RAM).

We further assume there is a component repository specifying the available components according to Definition 3. It also specifies composite components according to Definition 4 that predefine implementation patterns for the particular function blocks.

Definition 3 A *component* is specified by a tuple $(n, S_r, S_p, F_r, R_r, T, A_T, D_T)$ with the maximum number n of instantiations per processing node, the required services S_r, the provided services S_p, the required hardware capabilities F_r, the required amount of resources R_r, a set of tasks T describing the internal behaviour, precedence relations A_T between tasks, and the delegations D_T between the service interfaces $S_r \cup S_p$ and the tasks T. The required and provided services S_r and S_p are modelled by their interface type and the maximum number of clients/connections for each provided service.

Definition 4 A *composite component* is specified by $(t, S_r, S_p, C, A_C, D_C)$ with the type t (function, proxy, multiplexer or protocol stack), the required and provided services S_r and S_p, a set of internal non-composite components C, their assembly connections A_C and their delegations D_T to the external service interfaces S_r and S_p. In addition to their interface type (syntax), the required and provided services are modelled by their expected/provided function (semantics). The provided function is either statically defined or inherited from one of the composite's service requirements (in case of a proxy, multiplexer or protocol stack).

A composite component thus specifies a particular use of one or multiple components to achieve a certain function, proxy, multiplexer or protocol stack.

Result of the integration is a *component instantiation* according to Definition 5, which specifies what components must be executed on what processing unit and how the components' interfaces must be interconnected.

Definition 5 A *component instantiation* is a directed graph $G_{inst} = (V, A)$ in which the nodes $v \in V$ denote the components that are instantiated and directed edges $a \in A$ represent the service connections between components. An arc $a = (u, v)$ denotes that u connects to v. Nodes are annotated with the processing unit on which they are instantiated and the assigned resources (e.g. RAM, scheduling parameters). Edges are annotated with the service which they interconnect.

The component instantiation must not only adhere to the execution requirements of the components but also the latency requirements of the function architecture. It should also be minimal in the sense that, e.g. unused device drivers, should not be instantiated at all.

In order to perform a RTA to verify the latency requirements, we need a timing view in form of a task graph. Here, we only provide a very basic definition of a task graph. A more sophisticated model for microkernel-based systems has been presented in [24]. In our use case, tasks inherit their scheduling priority from the instantiated component from which they originate.

Definition 6 A *task graph* is a directed acyclic graph $G_{tasks} = (V, A)$ with tasks V and precedence relations A. Once activated, a task must execute for a certain amount of time to complete which is bounded by its best-case and worst-case execution time. A task that completed will immediately activate other tasks according to the specified precedence relations. Tasks without any incoming edge are further characterised by a lower and upper arrival curve that specify the minimum and maximum number of activations in a particular time interval.

4 Related work

There is a large body of research in the field of *model-based design and integration* for embedded systems. In particular, there are several languages and tools that capture model information and integrate model checking, analyses and/or code generation: EAST-ADL[1] is an established description language in the automotive domain and aligned with the classic AUTOSAR standard. Rubus [17] is a concept for the model-driven development of vehicular software systems which particularly expresses timing-related information to support end-to-end timing analysis. Similar to classic AUTOSAR, code generation is used in order to deploy the modelled software components into tasks and to create the corresponding communication primitives. AADL[2] is an approach similar to EAST-ADL that addresses other safety-critical domains such as avionics and aerospace. MARTE[3] addresses the modelling and analysis of embedded real-time systems by adding non-functional properties such as timing to UML. MechatronicUML [7] is a tool suite that integrates software- and control engineering for the development of mechatronic systems. Ptolemy II [22] is a software framework for studying the actor-based design of distributed CPS. It focuses on the assembly of software components and supports different models of computations, among which the Ptides model [32]. Furthermore, it supports aspect-orientation as a means to enrich a modelled design with additional concerns (e.g. contracts, execution timing, fault modelling) [1]. Lingua Franca [16] is a meta language for the definition and composition

[1] http://east-adl.info/.

[2] http://www.aadl.info/.

[3] https://www.omg.org/omgmarte/.

of reactors – a time-aware variant of the actor model – by reusing many ideas from Ptides. There are also approaches to combine existing tool suites into a consistent tool chain and workflow [5,28]. A good overview and characterisation of such multi-view modelling approaches is given by Persson et al. [20]. However, all these languages and tools are commonly geared towards visual or textual modelling that requires user interaction. Although they are very helpful in assisting the design and integration of embedded systems, design and integration decisions have to be performed by human engineers. Moreover, these tools often rely on code generation and a complete tool chain to statically implement and integrate the modelled applications as a whole.

In his position paper, Rushby [23] highlights the importance of self-integration as a means to provide new capabilities or services in a system-of-systems context and argues that it "requires automated verification and synthesis of monitors, adapters, and mediators at integration-time". Since 2014, the SISSY workshop [2] is dedicated to this problem statement (cf. [3]) that focuses on the self-integration of systems into larger system-of-systems rather than the automation of the integration phase of a single system.

A wide field of related work is known under the term *models@run.time* and addresses self-modelling aspects of all kind of systems to enable adaptivity. Most recently, a systematic literature review of 275 papers and a summary of research challenges have been presented by Bencomo et al. [4]. Our framework is an approach to combining existing runtime models and methods in order to achieve adaptability of heterogeneous platforms.

Another line of work can be summarised under the term *automated design-space exploration*. There are many approaches that try to formalise the design space in order to automatically perform architecture optimisation: Terzimehic et al. [29] propose Satisfiability Modulo Theories (SMT) for formulating constraints and objectives derived from hardware and software annotations. ProMARTES [30] is an approach that combines profiling, analysis and simulation for architecture performance optimisation using Genetic Algorithms (GAs). Eder et al. [9] use a dedicated domain-specific modelling language to formulate memory, safety, cost, energy and bandwidth constraints for the mapping/distribution problem, i.e. the allocation of software components to hardware components. The main limitation of these approaches is that the software composition is fixed. This is a valid assumption when considering the deployment of software components on a homogeneous architecture. However, as soon as software layers differ, a particular component might need to be integrated differently when deployed on another hardware component. There are approaches that explicitly address the software composition problem [21] using constraint-based methods such as SMT [21] or Mixed Integer Linear Programming (MILP) [14], however, they do not consider composite components. As soon as the number of required software components, tasks, etc. is not fixed before design-space exploration, formulating (system-level) constraints efficiently becomes very challenging. Furthermore, system-level effects and conflicts must be known and approximated a-priori to formulate those constraints in the first place. This is in contrast to our assumption that system-level effects can often only be determined by complex analyses. Although we do not regard constraint-based methods as holistic alternatives to our framework, we consider these as reasonable additions to solve subproblems efficiently.

Intermediate results of our model-based integration approach have been presented in [26] with a focus on an automotive use case. Yet, the preliminary work has not featured an overarching framework, a formalisation of cross-layer model operations, or a backtracking search algorithm.

5 Framework

Following our problem statement, the structure of the modelled architectural views is not predetermined. They must therefore be (automatically) constructed from the given input. This also involves the automation of integration decisions (e.g. picking an implementation variant) with potentially complex interdependencies. Under these conditions, formulating the entire solution space (e.g. to find a global optimum) quickly becomes unmanageable. With our framework, we provide a solution for this in which we follow a greedy strategy in combination with backtracking to deal with conflicts.

Our framework comprises four basic parts as illustrated by Fig. 4. The The cross-layer model (CLM) stores the modelled architectural views (which we call *model layers*) and captures their relations. It is incrementally built up and refined by executing operations of different types that we explain in Sect. 5.2. An operation restricts the type and the scope of modifications that are allowed on the CLM. The actual modifications are computed by analysis engines (AEs) that are invoked by the operations and which may use existing tools to solve the particular problem. The execution of operations is managed by a search algorithm as we will explain in Sect. 5.4. Our framework is applied to a particular integration problem by a) defining the layers and their hierarchy of the CLM, b) defining the operations (and their order) that shall be performed on the CLM, and c) implementing the corresponding AEs.

5.1 Cross-layer model

The CLM is composed of a fixed set of graph-based model layers $\mathbf{G} = \{G_1, \ldots, G_n\}$ and their relations $\mathbf{R}$ as explained

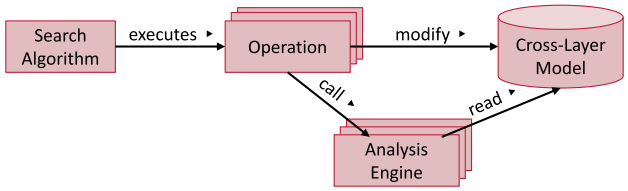

Fig. 4 Our framework comprises four basic parts

below. The nodes and edges of each model layer can be annotated with arbitrary parameters. Let us consider the function architecture from Fig. 3 as an example of such a model layer. For the distribution of function blocks on our processing platform from Fig. 2, we also want to annotate each function block with a discrete *mapping* parameter. For every block, we can choose from a candidate set that comprises node 1 and node 2 (or a subset thereof).

Definition 7 A *model layer* is a directed multigraph $G_k = (V_k, A_k, s_k, t_k)$ that is defined by a set of nodes V_k, a set of edges A_k and functions:

- $s_k : A_k \rightarrow V_k$ assigning each edge to its source node,
- $t_k : A_k \rightarrow V_k$ assigning each edge to its target node.

Every node and edge can be annotated with an arbitrary set of named parameters P. A parameter $p_i \in P$ is characterised by a set of candidates $cand(p_i)$ and a selected value $val(p_i) \in cand(p_i)$.

The CLM is built by capturing inter-layer relations between all pairs of layers: $\mathbf{R} = \{R_{i,j} | i < j : G_i, G_j \in \mathbf{G}\}$.

Definition 8 The inter-layer relation between two layers G_i and G_j is defined by a 3-tuple $R_{i,j} = R_{j,i} = (U_i, U_j, E_{i,j})$ with

- the set $U_i = V_i \cup A_i$ of nodes and edges from G_i,
- the set $U_j = V_j \cup A_j$ of nodes and edges from G_j,
- a relation $E_{i,j} \subseteq U_i \times U_j$.

5.2 Operations

As mentioned before, we restrict modifications on the CLM by our concept of operations. In particular, we distinguish four different operation types: *restrict* for modifying the candidate sets, *assign* for parameter decisions, *transform* for creating nodes/edges and setting inter-layer relations and *check* for performing admission tests. Operations are typically invoked separately for each node/edge of a model layer. However, in some cases, operations need a larger scope and thus require all nodes/edges of a model layer as input. We refer to the latter as *batch operations*.

5.2.1 Restrict

A *restrict* operation aggregates and constrains possible candidates of a particular parameter and model layer. It can be regarded as a layer-specific and parameter-specific function that defines the candidate set for a particular node/edge in order to define the search space. We denote this operation by $restrict_{k,i}(obj)$ and $batch\_restrict_{k,i}()$, where k identifies the model layer G_k, i the parameter p_i and obj the particular node/edge.

5.2.2 Assign

A *assign* operation selects one of the possible candidate values from the candidate set. It can be seen as a refinement step on the CLM in order to perform a depth-first search. As above, we denote this operation by $assign_{k,i}(obj)$ and $batch\_assign_{k,i}()$.

5.2.3 Transform

The purpose of a *transform* operation is to populate a particular model layer with nodes/edges by performing substitutions to the nodes/edges of an already populated model layer. We denote this operation by $transform_{k,l}(obj)$ and $batch\_transform_{k,l}()$, where k identifies the source model layer and l identifies the target model layer. While the batch operation can perform arbitrary substitutions, the non-batch variant must preserve locality in the sense that edges can only be created if they connect nodes that were transformed from interconnected nodes on the source layer. This is captured more formally by the following definition.

Definition 9 The *transform* operation substitutes a node/edge of a source layer G_k with a particular pattern (subgraph) that is inserted into a target layer G_l:

$$
\begin{aligned}
&transform_{k,l} : V_k \cup A_k \rightarrow \mathbb{G} \\
&\quad \text{if } x \in V_k : \quad x \mapsto (V, A, s, t) \\
&\quad \text{if } x \in A_k : \quad x \mapsto (V \cup V_s \cup V_t, A \cup A_s \cup A_t, s, t)
\end{aligned}
$$

with:

- $V_s = \{v : \exists (s(x), v) \in E_{k,l}\}$
- $V_t = \{v : \exists (t(x), v) \in E_{k,l}\}$
- $\forall a \in A : (s(a), t(a)) \in V \times V$
- $\forall a \in A_s : (s(a), t(a)) \in (V_s \times V) \cup (V \times V_s)$
- $\forall a \in A_t : (s(a), t(a)) \in (V_t \times V) \cup (V \times V_t)$

In the scope of our use case, common examples are the insertion of intermediate nodes (e.g. proxies) and the replacement of function blocks by one or multiple software

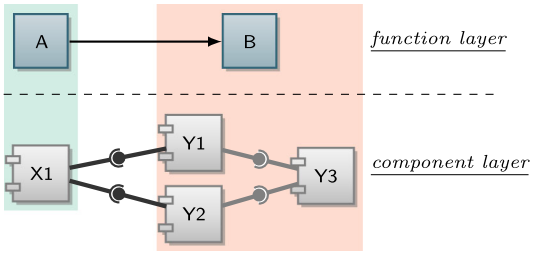

Fig. 5 Function blocks are transformed into components

components. An example is depicted by Fig. 5 that shows how two connected function blocks are translated into components. Block A is replaced by a single component X1, whereas block B is replaced by a pattern of three components (Y1,Y2 and Y3) and two edges. The edge between A and B is transformed into two edges that interconnect the component X1 ($\in V_s$) with Y1 and Y2 ($\in V_t$).

5.2.4 Check

The *check* operation is used for performing sanity checks and admission tests on a particular model layer. We denote this operation by $check_k(obj)$ and $batch\_check_k()$. For instance, we may check every node on a component layer whether its required services are connected. More sophisticated admission tests would be the check of latency and reliability requirements as mentioned in Sect. 3. If a check fails, the preceding parameter decisions must be revised to explore alternative combinations. This involves repeating the dependent operations. Ideally, only those decisions will be revised that actually influence the *check* operation.

5.3 Analysis engines

AEs are a practical approach to interface established methods, existing tools and to use model-, application-, and domain-specific logic for performing particular operations. Intuitively, an AE encapsulates expert knowledge and addresses a particular sub-problem, they perform local optimisation or implement heuristic decisions. AEs may access the CLM and the annotated parameters during its operation. Accesses to the CLM are tracked in order to construct a dependency graph that is used by the search algorithm that is explained in the following section.

5.4 Search algorithm

As mentioned before, we implement a depth-first search approach with our framework by selecting parameters early on. This may require iterations in case a check or assign operation fails. The latter is the case if there are no (more) candidate values to choose from. The search algorithm shall systematically explore alternative parameter candidates until the first feasible solution is found. While the AEs deal with prioritising what candidates are tried first (e.g. by implementing heuristic methods), the search algorithm must take care that a) all parameter combinations are reachable (complete search) and b) the algorithm terminates (i.e. combinations are not repeatedly tested). The central question of this algorithm is what operations to revert and re-execute if a particular *check* or *assign* fails. A straightforward approach is to apply a simple chronological backtracking algorithm that reverts the sequence of operations to the latest revisable operation. However, as the CLM and operations expose parameter decisions and restrict the scope of model modifications, we can also employ an automated tracking of dependencies between operations and thereby implement a non-chronological backtracking.

Listing 1 Search algoritm

```
# Input: list of templates                                    1
dg = DependencyGraph()                                        2
cm = CrossLayerModel()                                        3
remaining = Stack(reversed(templates))                        4
while op = remaining.pop():                                   5
    if not op.is_batch():                                     6
        for obj in op.layer.graph_objects():                  7
            if dg.has_node(op, obj):                          8
                continue                                      9
            cm.start_tracking                                 10
            okay = op.execute(obj)                            11
            rd, wr = cm.stop_tracking()                       12
            n = dg.insert_node(op, rd, wr, obj)               13
                                                              14
            if not okay:                                      15
                rollback(n)                                   16
    else:                                                     17
        cm.start_tracking()                                   18
        okay = op.execute():                                  19
        rd, wr = cm.stop_tracking()                           20
        n = dg.insert_node(op, rd, wr)                        21
        if not okay:                                          22
            rollback(n)                                       23
return cm                                                     24
                                                              25
rollback(node):                                               26
    dg.topological_sort(node)                                 27
    rn = dg.latest_revisable_node(node)                       28
    if not rn:                                                29
        throw "Not Found"                                     30
                                                              31
    for n in dg.reversed_subtree(rn, node):                   32
        cm.rollback(n)                                        33
        if n.op not in remaining:                             34
            remaining.push(n.op)                              35
        dg.remove(n)                                          36
```

The overall algorithm is listed as pseudo-code in Listing 1. A dependency graph (DG) takes care of tracking dependencies between operations (cf. Sect. 5.4.1). The template operations are iterated from first to last by consuming from the `remaining` stack (line 5). Non-batch operations are applied to every node and edge of the operation's asso-

ciated layer (line 7) unless they are already present in the DG for this node/edge (lines 8-9). When executing an operation, its read and write accesses to the CLM are tracked in order to insert a new node into the DG accordingly (lines 10-13). In case the operation was not successful (line 15), the DG is partially sorted (line 27, cf. Sect. 5.4.2) in order to define a rollback path. If there is a revisable decision (`rn`) on which the failed operation depends, the subtree of `rn` is iterated from the leaves upwards (line 32) in order to rollback the CLM (line 33), push the operation template back to the `remaining` stack (lines 34-35), and clean up the DG (line 36). For batch operations, the graph objects are not iterated and the operation is only executed once per layer (lines 18-23). In particular, all graph objects are passed to the operation at once and a single node is inserted into the DG.

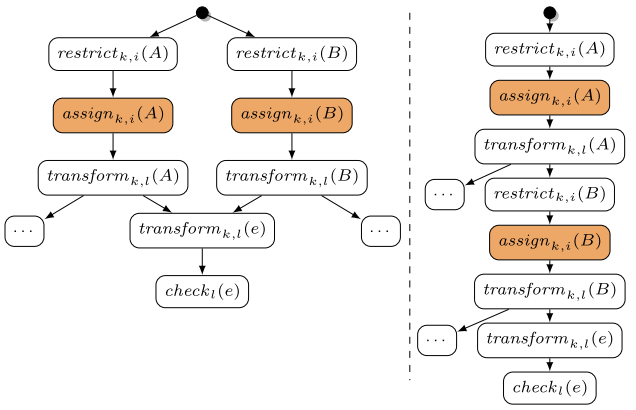

Fig. 6 **Left:** Exemplary dependency graph for Fig. 5. **Right:** Topologically sorted graph for failed check. Revisable operations are highlighted

5.4.1 Dependency tracking

The motivation of the dependency tracking is to allow non-chronological backtracking [31]. It is enabled by the fact that there is only a partial order between operations. For instance, in Fig. 5, it is undefined whether A is transformed before B or B transformed before A. In consequence, chronological backtracking will potentially revert operations that could otherwise be preserved between iterations. We implement non-chronological backtracking by tracking read and write accesses to the CLM for each executed operation. From this, we derive dependencies between operations as follows: An operation y depends on x if, e.g., y reads a parameter that is written by x. Note, that this also includes read and write accesses to the graph structure of the CLM. Dependencies are stored in a DG defined as follows.

Definition 10 The dependency graph is a directed acyclic graph $DG = (V, A)$ with

- a set of nodes V representing operations,
- a set of directed edges $A \subseteq V \times V$ such that $\forall(u, v) : \nexists x : (u, x), (x, v) \in A$.

The edges reflect the transitive reduction of execution dependencies between operations, i.e. an edge (u, v) denotes that v depends on u and (potentially) all operations on which u depends.

The left side of Fig. 6 depicts a simplified DG for the example from Fig. 5. In order to decide on the *implementation* parameter p_i for both function blocks on layer G_k, there is a $restrict_{k,i}$ and $assign_{k,i}$ operation on node A and B. As the *assign* operations read the candidate set written by the corresponding *restrict* operation, an edge exists between these operations. Similarly, the *transform* operation for A and B depend on the corresponding *assign* operation, as they access the selected parameter value. The *transform* operation of the edge e between A and B, however, depends on both node *transform* operations because it connects component nodes that were inserted (written) when the nodes where transformed. In this example, we assume there is a *check* operation that serves as a sanity check for the resulting component graph. It accesses the nodes and edges written by all three check operations, yet, only the dependency to the edge transformation is stored in the graph since it transitively includes the dependencies to the other *transform* operations. In consequence, the result of the *check* operation can be affected by any operation that is a predecessor in the DG. Next, we will explain how this DG enables the rollback of operations to systematically revise and iterate parameter decisions. Note, dependency tracking could also be used to determine the initial execution order of operations, which must currently be provided by the user.

5.4.2 Rollback and revision

Let us assume, the check operation in Fig. 6 fails and that both assign operations are revisable, i.e. there are alternative candidate values left. The goal is to systematically iterate all combinations of possible values for both parameters. For this, we must keep in mind that the set of parameters and possible candidate values is neither predetermined nor globally fixed but depends on other operations (e.g. the number of inserted components). In order to iterate parameter alternatives in a systematic pattern and still respect dependencies between operations, we take the following approach: First, we store the already tried and discarded candidates for each parameter in a local context, i.e. for the corresponding assign operation, so that it is reset when the operation is reverted. Second, when backtracking is triggered, we sort the predecessors of the failed operation topologically in order to define an unambiguous path in which the revisable operations can be hierarchically iterated. This is exemplified by the right

side of Fig. 6 that shows the topologically sorted DG from Fig. 6. In the new graph, the preceding operations have been sequentialised such that $assign_{k,i}(A)$ is before $assign_{k,i}(B)$ while independent sub graphs (indicated by "…") remain unmodified. The rollback procedure on this graph works as follows: First, the current parameter value for B is stored in the set of discarded candidates, every operation in the subtree of $assign_{k,i}(B)$ is reverted and also removed from the DG (Listing 1, lines 33+36). Second, the assign operation is revised, i.e. re-executed but disallowing the already discarded values. If no other candidate is left, the operation fails and the algorithm will jump to $assign_{k,i}(A)$. All candidates for B will be iterated again for a new value for A. There is, however, a special case that we must take into account: Later operations may transitively add dependencies to operations that have already been sorted. For instance, such an operation may depend on $transform_{k,l}(e)$ and a node from a previously independent sub graph. If the new operation fails, its predecessors must be sorted again. Due to the fact that topological sorting is not unambiguous, the order must be preserved between multiple sort operations. We resolve the possible ambiguity by preferring operations from earlier iterations when sorting. This way, the later added operation will always be appended to the already sorted path.

6 Implementation and evaluation

In this section, we apply the presented framework to our use case and provide an evaluation. We implemented the framework in the Python programming language. We used the module *networkx*[4] for storing directed multigraphs whose nodes and edges can be annotated with labelled attributes that we use for storing the particular parameter values and candidates. The source code is available online[5].

6.1 Framework application

Please note, that we cannot provide the full details on how our framework is applied to the use case in the scope of this paper. We therefore provide a summary by going through the most relevant operations and model layers with emphasis on the function block *preprocess* (cf. Fig. 3). We randomised most of the parameter decisions for evaluating the search algorithm and omit tailoring the AEs to our use case.

First, a distribution of function blocks to compatible processing nodes is defined by selecting (restrict and assign) the *mapping* parameter of all nodes. This is performed as a batch operation in order to have a global view on the function architecture and to prefer parameter combinations that minimize the number of connections between blocks mapped to different nodes. There are four implementation options for the preprocess block of which two require the FPGA that is only available on node 1, hence the latter will be preferred over node 2. Furthermore, the compute disparity block has a hardware (FPGA) implementation and a software implementation whereas there are two software implementations for computing the pointcloud. W.r.t. compatibility constraints, the object detection and pose estimation blocks shall always execute on node 2 while the camera block is only compatible to node 1.

Depending on the mapping, communication proxies must be inserted between dependent but distributed blocks. Hence, a *proxy* parameter is selected for all edges. It determines whether and what communication proxy must be inserted to establish a communication path between the connected function blocks. As there is only a single network in this use case, there is only a single candidate for every edge that, however, depends on the mapping. A transform operation reflects the new structure including proxies in the communication architecture layer by copying the nodes and adding an intermediate proxy node to the edges with a set proxy parameter. On this layer, the *implementation* parameter is selected randomly in order to select one of the available and compatible implementations for each function block.

Depending on the *implementation* parameter, the preprocess block is transformed into a network of 2, 3 or 4 components. One of the components requires a service from the camera and another (or the same) component provides a service delivering the preprocessed image. The edges of this layer are transformed so that they connect the corresponding components. As the connected components must respect service compatibility as well as cardinality restrictions (i.e. maximum number of clients), there are two more steps before we have a correct component architecture. First, a *protocol-stack* parameter is selected for every edge which determines whether and what protocol stack component must be inserted to achieve service compatibility. An intermediate model layer is populated by a transform operation that copies the component nodes and inserts the protocol stack components for the edges according to the previously selected parameter. Similarly, a *muxer* parameter is selected for every node on the new layer to determine if resource multiplexers need to be inserted and to set up the transformation into the component architecture layer. On this layer, we select an *instance* parameter for every node to determine whether duplicate components can be combined into a single component instantiation. The transformation into the component instantiation layer may insert the same object for multiple nodes, which is a valid operation in our framework.

At this point, we can check whether memory requirements of all component on the same processing can be met.

[4] http://networkx.github.io/.

[5] https://github.com/IDATUBS/MCC.

In this use case, we employ symmetric multiprocessing on node 1. To support latency analysis, we want to predetermine on what core and with what scheduling priority a software component will be executed. We therefore select an *affinity* and a *priority* parameter for every node on the component instantiation layer. While we choose the *affinity* for every instance uniformly at random, the *priority* is assigned as a batch operation. We only consider a single priority assignment as established (optimal) priority-assignment schemes [6] should be applied that prevent iterating all priority permutations. Every component is represented by one or multiple tasks that can be triggered by interactions with other components via the service interfaces, which is specified in the component repository. This allows transforming the component instantiation layer into a task graph layer. The task graph can be used to calculate the reliability of critical processing chains for a given error rate and to validate whether the reliability requirements are met. As the FPGA is more susceptible to particle flux than the CPU, we assume that processing chains that use the FPGA will achieve lower reliability than software implementations. For the latency analysis, we applied the analysis from [24] that serves as a basis for the last check operation.

6.2 Evaluation

In this section, we evaluate the application of our framework to the use case. In particular, we show a) how our frameworks performs for finding a first feasible solution for a given input and b) how our framework reacts adaptively to changes of model parameters. For a), we compare different variants of our use case and compare chronological with non-chronological backtracking. For b), we trigger adaptations after a solution was found by randomly increasing the Worst-Case Execution Time (WCET) assumptions for our component tasks.

For both function architectures depicted in Fig. 3, i.e. pose estimation (POSE) and object detection (OBJ), we consider three variants:

I. FPGA disabled, low reliability required
II. FPGA enabled, low reliability required
III. FPGA enabled, high reliability required

Note, that we synthesised the execution time values for the components' tasks (i.e. their internal behaviour) because not all execution traces for the implementation variants were available. More specifically, we categorised the tasks of each component into small tasks (100 us), medium tasks (1 ms) and large tasks (20 ms). We also added some background load to both processing nodes for realistic operating conditions rendering the integration process more challenging w.r.t. the latency requirement.

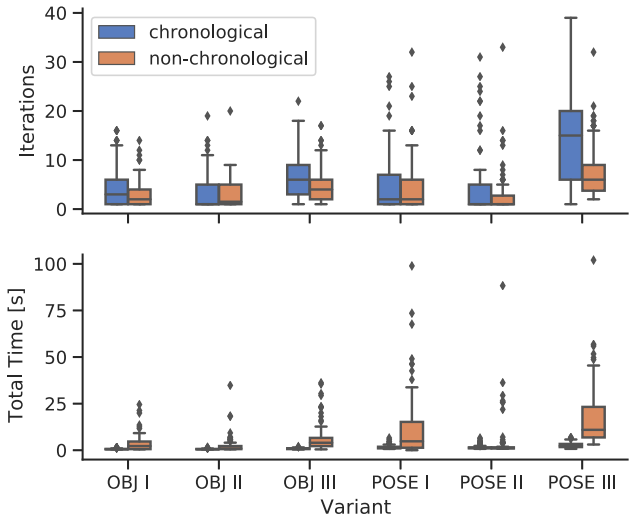

Fig. 7 Compare required iterations and time for chronological and non-chronological backtracking (each box is calculated from 100 samples)

Table 1 summarises the resulting characteristics of our use case variants that we acquired by a full sweep of the search space. It displays the number of feasible solutions. For every solution, the minimum and maximum number of variables (i.e. parameters with more than one candidate) is stated. We also denote the possible combinations of parameter assignments (assuming independent variables). The last column indicates a lower bound on the required operations which is determined by the minimum/maximum number of nodes contained in the dependency graphs of the solutions. Note, that not all possible combinations needed to be iterated for the full sweep as variables are often dependent: For OBJ, 31 candidate solutions were rejected by the latency analysis. Another 8 solution candidates were rejected earlier due to inconsistencies in the distribution (missing protocol stack). In OBJ III, 4 additional solutions were rejected by the reliability check. For POSE I and III, latency analysis rejected 206 solutions (211 for POSE II). Reliability analysis rejected 16 solutions for POSE III. Again, because of inconsistencies 16 solutions were rejected early in POSE I-III.

In order to evaluate the search efficiency, we executed the search multiple times and recorded the number of iterations required to find the first solution and the processing time (executed on a single core of an Intel i5-3210M @ 2.50 GHz). Figure 7 depicts the comparison between chronological and non-chronological backtracking as Tukey box plots. Although we randomised the affinity and implementation decision, only a few iterations (in comparison to the number of possible combinations) are required on average to find a feasible solution. As expected, non-chronological backtracking requires fewer iterations. This becomes particularly evident in OBJ III and POSE III, for which the reliability requirement leads to late rejections. As the reli-

Table 1 Characteristics of the six use case variants

Variant	Solutions	Variables	Possible combinations	Required operations
OBJ I	9	6–8	64–256	881–973
OBJ II	13	4–8	32–512	881–973
OBJ III	9	6–8	128–512	881–973
POSE I	50	10–11	1024–2048	1097–1147
POSE II	61	7–12	128–4096	1070–1147
POSE III	50	11–12	2048–4096	1097–1147

Table 2 Number of successful adaptations for every variant and percentage value

	OBJ I	OBJ II	OBJ III	POSE I	POSE II	POSE III
10%	66	2	48	25	33	21
50%	18	20	3	14	8	16
100%	8	8	2	11	28	3

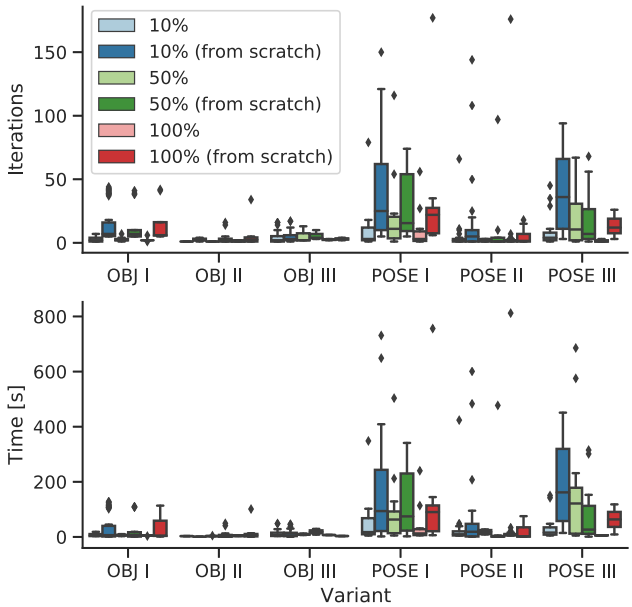

Fig. 8 Required iterations and time for our adaptation scenarios

ability test does not depend on the affinity parameters, the non-chronological backtracking can cut some combinations off the search space where the chronological backtracking will iterate alternative affinity decisions before changing the mapping parameter. On the other hand, the processing time obviously increases for the non-chronological backtracking due to the additional bookkeeping of the dependency graph. In consequence, chronological backtracking will be faster if the iterations are inexpensive. However, in case of more expensive operations (e.g. complex admission tests), non-chronological backtracking can be beneficial. Note, that in three samples, the latency analysis did not converge,[6] which led to extreme outliers for the processing time. For better readability, these samples are not shown in the box plot for the processing time.

For the second part of our evaluation, we investigate whether our framework can be applied to achieve self-adaptability and self-reflection. When combined with runtime monitoring, model deviations can be detected and fed back into our framework as a parameter change. Because the DG exposes what operations are affected by the parameter change, the search algorithm is able to rollback and re-execute these operations. For low-level parameters, mainly check operations will be affected. If these check operations fail, the search must be continued to find a solution that can deal with the changed parameter. For our evaluation, we simulated WCETs adaptations of arbitrary tasks by the following experiment. For every found solution, we randomly pick a task from the task graph and increase its WCET parameter by a predefined percentage. The framework then rolls back all dependent operations. In this case, only the latency analysis is rolled back and re-executed to validate whether the changed parameter affects this admission test. If it does, the exploration is continued by rolling back to the nearest revisable decision in the DG to find a solution for the adaptation. This is repeated until there are no further solutions, i.e. the framework cannot deal with the adaptation any more. We performed this experiment for all previously mentioned variants of our use case with three different percentage values (10%, 50% and 100% increase of WCETs). Table 2 shows the number of successful adaptations. As expected, a 10% increase in WCETs does not affect the schedulability as much as a 50% or 100% increase, hence there are more adaptations possible on average. However, as the tasks for which to increase the WCET are picked randomly for every experiment, the results are prone to stochastic effects. The higher the priority, the larger the WCET, and the higher the activation rate of a task, the more impact a WCET increase will have on the schedulability. The number of successful adaptations thus also depends on when such a sensitive task was picked in the

[6] This is a known effect of advanced timing analysis using network calculus or busy-window analysis. It indicates low robustness (w.r.t. timing) of the investigated system. In practice, such cases would be dropped after bounded analysis time.

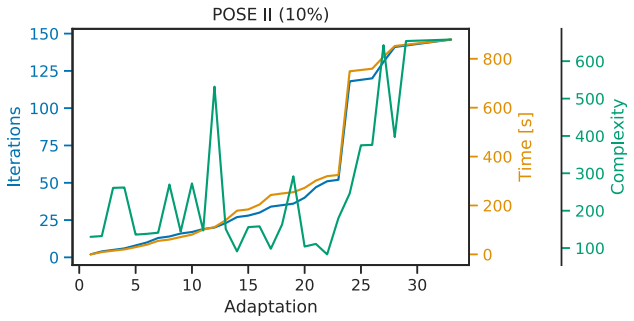

Fig. 9 Development of total iterations, time and complexity

experiment. Every adaptation serves as a datapoint in Fig. 8 which summarises the required iterations and processing time for every adaption as Tukey box plots. For comparison, the figure also shows the results when every parameter adaptation starts a search from scratch (which requires significantly more time and iterations in general). For one case (POSE III, 50%), searching from scratch was more efficient. This is explained by the fact that the adaptation experiment performs a local search, which is dictated by the very first solution. Figure 9 depicts the development of the total number of iterations, the total processing time and the search space complexity (number of total iterations + combinations left to iterate) for every adaptation for POSE II and 10% increase as an example. Although the charts look differently for the other variants, this case emphasises our observations. First, the required processing time correlates with the number of iterations. Second, the slope of both curves indicate how far the search algorithm needed to rollback the model and extend the search scope. Third, there are multiple spikes in the complexity since the search space is not holistically defined in the beginning but extended iteratively. However, instead of iterating all newly generated combinations, the framework is able to cut-off parts of the search space because of detected conflicts.

7 Conclusion

In this paper, we presented a framework that enables modelling and automating integration activities. The framework builds upon a cross-layer model that captures arbitrary architectural views and their relations. Integration activities are represented as incremental operations on this model including automated decision-making. Instead of a holistic (and potentially approximated) formulation of decisions and constraints, analysis engines perform decisions and admission tests on demand. In particular, this allows the integration of existing methods and specialised tools that implement these admission tests. On the one hand, admission tests provide automated verification of functional and non-functional requirements. On the other hand, they identify emergent constraints that were not previously known. We pursued a depth-first search approach to iterate alternative decisions only if admission tests fail.

Our framework addresses two challenges in particular: (a) Automated composition of a system from components and synthesising architectural views that can be passed to model-based verification tools. (b) Tracking of dependencies between model operations so that the scope of parameter changes can be limited. Dependencies are captured conservatively based on accesses to the cross-layer model. This traceability is not only essential for enabling a non-chronological backtracking during integration. It also builds the foundation for incremental adaptations and self-reflection in reaction to unforeseen changes/anomalies observed at runtime (by monitoring).

In our use case, we demonstrated the applicability of this framework in conjunction with a component-based design and a component-based OS. In particular, we could show that the framework can be used for combining heuristic decision-making with late admission tests (latency, reliability) that perform an automated verification.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Akkaya I, Derler P, Emoto S, Lee EA (2016) Systems engineering for industrial cyber-physical systems using aspects. Proc IEEE 104(5):997–1012
2. Bellman K, Botev J, Diaconescu A, Esterle L, Gruhl C, Landauer C, Lewis PR, Stein A, Tomforde S, Würtz RP (2018) Self-improving system integration - status and challenges after five years of sissy. In: 2018 IEEE 3rd international workshops on foundations and applications of self* systems (FAS*W), pp 160–167
3. Bellman K, Gruhl C, Landauer C, Tomforde S (2019) Self-improving system integration—on a definition and characteristics of the challenge, pp 1–3. https://doi.org/10.1109/FAS-W.2019.00014
4. Bencomo N, Götz S, Song H (2019) Models@run.time: a guided tour of the state of the art and research challenges. Softw Syst Model 18:10

5. Biehl M, El-Khoury J, Loiret F, Törngren M (2014) On the modeling and generation of service-oriented tool chains. Softw Syst Model 13(2):461–480
6. Davis RI, Cucu-Grosjean L, Bertogna M, Burns A (2016) A review of priority assignment in real-time systems. J Syst Archit 65:64–82
7. Dziwok S, Pohlmann U, Piskachev G, Schubert D, Thiele S, Gerking C (2016) The MechatronicUML design method: process and language for platform-independent modeling. Tech. rep., Software Engineering Department, Fraunhofer IEM/Software Engineering Group. Heinz Nixdorf Institute, Paderborn, Germany
8. Dörflinger A, Albers M, Fiethe B, Michalik H, Möstl M, Schlatow J, Ernst R (2019) Demonstrating controlled change for autonomous space vehicles. In: NASA/ESA conference on adaptive hardware and systems (AHS)
9. Eder J, Zverlov S, Voss S, Khalil M, Ipatiov A (2017) Bringing dse to life: Exploring the design space of an industrial automotive use case. In: ACM/IEEE 20th international conference on model driven engineering languages and systems (MODELS)
10. Feske N (2020) Genode OS Framework Foundations 20.05. Tech rep
11. Hamad M, Schlatow J, Prevelakis V, Ernst R (2016) A communication framework for distributed access control in microkernel-based systems. In: Annual workshop on operating systems platforms for embedded real-time applications (OSPERT)
12. Hamzah RA, Ibrahim H (2016) Literature survey on stereo vision disparity map algorithms. J Sens 2016:8742920. https://doi.org/10.1155/2016/8742920
13. Härtig H (2002) Security architectures revisited. In: 10th ACM SIGOPS European Workshop. ACM, New York
14. Kirov D, Nuzzo P, Passerone R, Sangiovanni-Vincentelli AL (2017) Archex: An extensible framework for the exploration of cyber-physical system architectures. In: Design automation conference. ACM
15. Kuz I, Liu Y, Gorton I, Heiser G (2007) CAmkES: a component model for secure microkernel-based embedded systems. J Syst Softw 80(5):687–699
16. Lohstroh M, Romeo Í.Í, Goens A, Derler P, Castrillon J, Lee EA, Sangiovanni-Vincentelli A (2019) Reactors: a deterministic model for composable reactive systems. In: Cyber physical systems. Model-based design. Springer, Berlin, pp 59–85
17. Mubeen S, Mäki-Turja J, Sjödin M (2014) Communications-oriented development of component-based vehicular distributed real-time embedded systems. J Syst Architect 60(2):207–220. https://doi.org/10.1016/j.sysarc.2013.10.008
18. Möstl M, Nolte M, Schlatow J, Ernst R (2019) Controlling concurrent change—a multiview approach toward updatable vehicle automation systems. In: Saidi S, Ernst R, Dirk Ziegenbein E (eds) Workshop on autonomous systems design (ASD 2019), OpenAccess Series in Informatics (OASIcs), vol 68, pp 4:1–4:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Florence, Italy. http://drops.dagstuhl.de/opus/volltexte/2019/10337/
19. Möstl M, Schlatow J, Ernst R, Dutt N, Nassar A, Rahmani A, Kurdahi FJ, Wild T, Sadighi A, Herkersdorf A (2018) Platform-Centric Self-Awareness as a key enabler for controlling changes in CPS. In: Proceedings of the IEEE, vol 106
20. Persson M, Törngren M, Qamar A, Westman J, Biehl M, Tripakis S, Vangheluwe H, Denil J (2013) A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems. In: Proceedings of the eleventh ACM international conference on embedded software (EMSOFT)
21. Peter S, Givargis T (2015) Component-based synthesis of embedded systems using satisfiability modulo theories. ACM Trans Des Autom Electron Syst 20(4):49
22. Ptolemaeus C (2014) System design, modeling, and simulation: using Ptolemy II, vol 1. Ptolemy.org, Berkeley
23. Rushby J (2016) Trustworthy self-integrating systems. Distributed Computing and Internet Technology. Springer, Berlin
24. Schlatow J, Ernst R (2017) Response-time analysis for task chains with complex precedence and blocking relations. In: ACM Transactions on Embedded Computing Systems ESWEEK Special Issue (2017)
25. Schlatow J, Möstl M, Ernst R (2019) Self-aware scheduling for mixed-criticality component-based systems. In: Real-Time and Embedded Technology and Applications Symposium (RTAS)
26. Schlatow J, Nolte M, Möstl M, Jatzkowski I, Ernst R, Maurer M (2017) Towards model-based integration of component-based automotive software systems. In: Annual conference of the IEEE industrial electronics society (IECON17), Beijing, China. https://doi.org/10.24355/dbbs.084-201803221525
27. Song J, Wang Q, Parmer G (2013) The state of composite. In: Workshop on operating systems platforms for embedded real-time applications (OSPERT)
28. Sztipanovits J, Bapty T, Neema S, Howard L, Jackson E (2014) OpenMETA: a model- and component-based design tool chain for cyber-physical systems. Springer, Berlin
29. Terzimehic T, Voss S, Wenger M (2018). Using design space exploration to calculate deployment configurations of IEC 61499-based systems. In: IEEE international conference on automation science and engineering, CASE, Munich, Germany
30. Triantafyllidis K, Aslam W, Bondarev E, Lukkien JJ, de With PH (2016) ProMARTES: accurate network and computation delay prediction for component-based distributed systems. J Syst Softw 117:10
31. van Beek P (2006) Chapter 4: Backtracking search algorithms. In: Rossi F, van Beek P, Walsh T (eds) Handbook of constraint programming, vol 2. Elsevier, New York
32. Zhao Y, Liu J, Lee EA (2007) A programming model for time-synchronized distributed real-time systems. In: 13th IEEE real time and embedded technology and applications symposium (RTAS'07). IEEE, pp 259–268

Johannes Schlatow received the M.Sc. degree in computer and communication systems engineering from Technische Universität Braunschweig, Braunschweig, Germany, in 2013. Afterwards, he joined the Embedded System Design Automation Group of Prof. Ernst, where he conducted research in the field of design, modeling and analysis of component-based mixed-critical systems, and where he received his Dr.-Ing degree in electrical engineering in 2021. He is currently employed at Genode Labs GmbH, where he puts component-based operating systems into practice.

Edgard Schmidt received the B.Sc. degree in computer science from Technische Universität Braunschweig, Braunschweig, Germany, in 2020. He is a software developer and works at Sternico on Siemens Mobility projects. As a free software enthusiast and passionate programmer since his youth, his interests centre on software engineering and the impact of software on everydays life.

Rolf Ernst received the Diploma degree in computer science and the Dr.Ing. degree in electrical engineering from the University of Erlangen-Nuremberg, Erlangen, Germany, in 1981 and 1987, respectively. After two years with Bell Laboratories, Allentown, PA, USA, he joined the Technische Universitaet Braunschweig, Braunschweig, Germany, as a Professor of Electrical Engineering. He chairs the Institute of Computer and Network Engineering (IDA) covering embedded systems research from computer architecture and realtime systems theory to challenging automotive, aerospace, or smart building applications. Prof. Ernst is a DATE Fellow. He is a member of the German Academy of Science and Engineering (acatech). In 2014, he received the annual Achievement Award of the European Design Automation Association (EDAA).