

Synthesis-based Optimization and Verifi- cation of Component- based Systems

TNO-2026-17823 – 2 September 2026

Synthesis-based Optimization and Verification of Component-based Systems

Author(s)	Benny Akesson, Ben Pronk, Debjyoti Bera, Sven Weiss, and David Worrell
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	49 (excl. front and back cover)
Number of appendices	0
Sponsor	Hans Schurer
Project name	TechFlex 2025
Project number	060.63189/01.01

All rights reserved

No part of this publication may be reproduced and/or published by print, photoprint, microfilm or any other means without the previous written consent of TNO.

Disclaimer

In the creation of this document, Microsoft 365 Copilot was used to improve the use of English. In all instances where Copilot was used, the authors thoroughly reviewed the results to ensure the end result is accurate and free from errors, thus allowing the authors to take responsibility for the content.

Acknowledgement

The research is carried out as part of the TechFlex program under the responsibility of TNO-ESI in cooperation with Thales. The research activities are co-funded by Holland High Tech | TKI HSTM via the PPP Innovation Scheme (PPP-I) for public-private partnerships.

© 2026 TNO

Summary

Platform-based design enables high-tech equipment manufacturers to deliver many product variants by configuring reusable hardware/software components, but growing system diversity and long product lifecycles make integration and evolution increasingly costly. Engineers must select and integrate hundreds of components, reason about trade-offs (e.g., performance vs. cost), and repeatedly re-verify functional and non-functional requirements after platform updates. This challenge cannot be solved by adding more engineers, as there is also a growing scarcity of skilled labor.

The TechFlex 2025 project investigates how to move from expert-driven, manual configuration and late-stage validation to model-based automation for component-based product families. Concretely, it addresses three research questions: **(RQ1)** to what extent is it possible to synthesize software configurations from a product (platform) variation tree that satisfy functional requirements, constraints, and performance requirements while optimizing other system qualities such as cost? **(RQ2)** what defines a (minimal) set of component behaviors and interface specifications needed to generate an executable system model that can be used to validate system flows and performance early, without requiring a complete model of all components? **(RQ3)** to what extent is it possible to accurately predict the performance of arbitrary product configurations based on incremental insights for individual components gained during development and deployment?

As a result of research, the project has delivered the following three key contributions:

- **Automatic synthesis of product variants based on functionality and constraints (RQ1):** A model-based synthesis methodology that enables synthesis of hierarchically decomposed software architectures deployed on complex hardware configurations is proposed. It contributes generalized meta-models for software and hardware, as well as a Python-based toolchain for design-space exploration. Evaluation using both synthetic and industrial-scale cases indicates feasible runtimes (seconds to minutes) and valid synthesized solutions.
- **Component specification required for early system performance analysis (RQ2):** A methodology for generating executable models (mock components) from ComMA interface and component specifications to enable early platform verification (including timing constraints) prior to full software availability. It identifies limitations of reusing trace-based component semantics for code generation and proposes run-to-completion restrictions plus an operational interpretation that supports synthesis. The approach is demonstrated on a small example, showing how generated mocks can be deployed to detect and diagnose timing issues and platform-induced regressions.
- **Incremental performance prediction for product families (RQ3):** A compositional strategy for predicting performance of arbitrary deployments from independent component characterizations, focusing on shared-resource interference, is proposed. These independent characterizations can be combined to predict performance degradation for any pair of co-deployed components sharing memory without exhaustive co-deployment experiments, with good predictive accuracy.

Contents

Summary	3
Abbreviations	5
1 Introduction	6
2 System Synthesis	9
2.1 Introduction to System Synthesis.....	9
2.2 Requirements and Scope	10
2.3 State-of-the-Art.....	11
2.4 Choosing a Starting Point.....	14
2.5 Modelling Hierarchical Systems.....	15
2.6 Synthesis of Hierarchical Systems.....	19
2.7 Evaluation.....	24
3 Synthesis of Mock Components using ComMA	28
3.1 Introduction.....	28
3.2 State-of-the-Art.....	29
3.3 Model-based Mocking with ComMA	30
3.4 Application to the Meal Delivery System.....	39
4 Conclusions	44
4.1 Answers to the Research Questions.....	44
4.2 Future Work.....	46
References	47

Abbreviations

Abbreviation	Meaning
AADL	Architecture Analysis & Design Language
ALF	Action Language for Foundational UML
ComMA	Component Modeling and Analysis
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DEAP	Distributed Evolutionary Algorithms in Python
DSL	Domain-Specific Language
fUML	Foundational UML
GDP	Gross Domestic Product
GPU	Graphics Processing Unit
ILP	Integer Linear Programming
JSON	JavaScript Object Notation
OSGi	Open Services Gateway Initiative
PPP-I	Public-Private Partnership Innovation Scheme
QoS	Quality of Service
QRM	Quality and Resource Management
QRML	Quality and Resource Modeling Language
R&D	Research and Development
RQ	Research Question
SMT	Satisfiability Modulo Theories
Sol	Source of Interference
SSP	Structural Synthesis of Programs
SysML	Systems Modeling Language
TKI HTSM	Top Consortium for Knowledge and Innovation, High Tech Systems & Materials
UML	Unified Modeling Language
XML	eXtensible Markup Language

1 Introduction

The Dutch high-tech equipment industry, including companies like Philips, ASML, Canon Production Printers, and Thales, is a cornerstone of the Dutch economy, contributing €72 billion (6%) to GDP each year and accounting for about 4% of jobs. With 40% of private R&D spending, the industry is a key driver behind the innovative power of the Netherlands [1]. However, the industry is challenged by increasing system complexity, driven by a number of technological and market trends [2]. One example of such a trend is *growing system diversity*. Every customer wants a system tailored to their specific needs, which results in a growing number of unique system configurations, referred to as *product variants*, that must be developed and maintained. Individually developing and maintaining a large number of product variants is both costly and time-consuming. To address this, the industry has moved towards *platform-based design* [3]. This allows new product variants to be developed by configuring a common product platform, selecting hardware and software components from a repository of reusable assets.

A key characteristic of high-tech equipment is that systems have very long lifetimes, often several decades [1]. During this time, customers expect their systems to *continuously evolve* to counter hardware and software obsolescence, stay secure, and frequently provide new value-added features. It is hence important that the system is designed in such a way that the hardware and software platform can be easily changed as technology evolves and that it is possible to verify that both functional and non-functional (e.g. timing) requirements are still satisfied after an update or upgrade with limited effort.

While significant progress has been made to address challenges related to system diversity and evolution over the past years, there are still several important challenges to overcome as the industry is challenged to increase R&D productivity, while facing growing scarcity of skilled engineers [2].

C1) Specifying the software components to integrate to realize a new product variant in the best way requires considerable expertise: A software integrator working with complex high-tech equipment needs to be familiar with hundreds of components and how they fit together. There may be more than one set of components that can satisfy a given set of requirements, but the options are difficult to oversee, and the impact of design decisions is hard to relate to key system qualities such as performance or cost. There is therefore a need for *democratization of engineering* to address the scarcity of experienced engineers, as well as *optimization* to improve the competitiveness of the resulting systems.

C2) Software component integration takes a lot of time and effort: There is a need for *automation* to increase engineering productivity.

C3) Updating and verifying functional and non-functional requirements of product variants as product platforms and variants evolve is costly and time-consuming: There is a need for technology-agnostic *abstractions* and *automation* to reduce the effort involved in updating components as software technology evolves and verifying that requirements are still satisfied.

This report describes the results from the TechFlex 2025 project. TechFlex 2025 addresses the mentioned challenges associated with system diversity and evolution at the level of the software platform, building on results from TechFlex 2024 [3]. Specifically, the project investigates the following three research questions:

RQ1) [Automatic synthesis of product variants based on functionality and constraints]

- To what extent is it possible to synthesize from a product (platform) variation tree, software configurations that satisfy both functionality requirements, constraints, and performance requirements, while optimizing other system qualities such as cost?

RQ2) [Component specification is required for early system performance analysis]

- What defines a (minimal) set of component behaviors and interface specifications needed to generate an executable system model that can be used to validate system flows and their performance in an early phase without the need for a complete model of all components?

RQ3) [Incremental performance prediction for product families]

- To what extent is it possible to accurately predict performance of arbitrary product configurations based on incremental insights for individual components gained during development and deployment?

This report primarily addresses the first two research questions. The third research question is discussed in a master's thesis conducted in connection with this project. The high-level conclusions from this thesis are presented in Chapter 4 of this report. The research for RQ3 builds on an earlier thesis about performance prediction, conducted as part of the earlier ArchViews project, and described in [5].

The result of the overarching TechFlex project is a methodology for model-based specification, deployment, and verification, illustrated in Figure 1. The two first contributions of this methodology are results from TechFlex 2024 [3] and are considered context in this report, while the last two contributions are from TechFlex 2025:

1. **Technology-agnostic specification** of reusable building blocks (an abstraction)
 - Specification of product variants using a family of Domain-Specific Languages (DSLs), minimizing manual effort
2. **Technology-specific generation** of build, packaging, and deployment artifacts (automation)
 - Generators are phased in and out as software technologies evolve, improving portability of the whole product family
3. **Synthesis of component compositions and deployment from a high-level component specification** (democratization, automation, optimization)
 - Automatic system composition and optimization that increases product quality and reduces the dependency on scarce experts
4. **Platform verification** using an executable model generated from a high-level component specification (automation)
 - Early integration testing (shift left) on relevant hardware and software platforms, reducing the verification effort

The four contributions of this methodology address the specified challenges, although not through a simple one-to-one relation. For example, Challenge C1 is primarily addressed through synthesis (the third contribution), but technology-agnostic specifications (the first contribution) are required as input for synthesis. To actually realize the synthesized system,

we need the technology-specific generators (the second contribution) to automatically generate the relevant development artifacts. This provides automation that also addresses Challenge C2. Challenge C2 is also addressed by the fourth contribution, which allows generation of an executable model (mock component) from the technology-agnostic specification, enabling early integration testing on actual hardware before all components have been implemented. For Challenge C3, the distinction between technology-agnostic specifications and technology-specific generators is helpful because it allows us to focus our efforts on modifying or creating new technology-specific generators as technologies evolve.

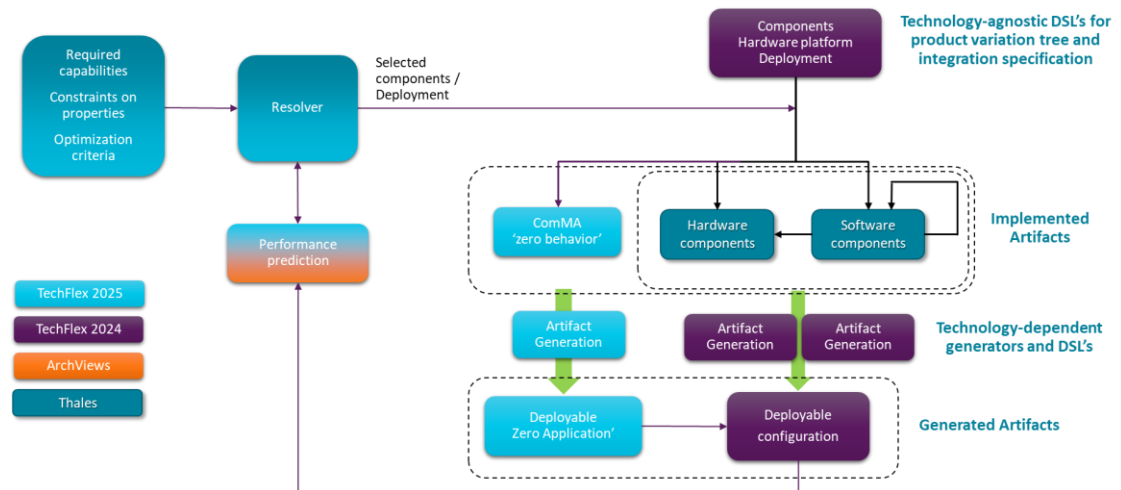


Figure 1: Overview of the TechFlex methodology

The remainder of this report is structured as follows. Chapter 2 addresses RQ1 and introduces a model-based methodology for automatic synthesis of component compositions to instantiate new product variants that satisfy both functional and non-functional requirements. Chapter 3 considers RQ2 and presents a methodology for early verification of the hardware and software platform in the context of such synthesized variants based on an executable model that is automatically generated from high-level interface and component (ComMA) models. Lastly, Chapter 4 rounds off the report with conclusions and future work.

2 System Synthesis

After stating the challenges and research questions addressed in this work in Chapter 0, we proceed in this chapter by describing how some of the stated challenges can be addressed through automatic system synthesis, or more specifically, synthesis of software component compositions and deployments. First, in Section 2.1, we provide an introduction to this topic. Section 2.2 proceeds by discussing the requirements and scope of our research, after which Section 2.3 provides a brief review of the state-of-the-art to identify key concepts and research gaps. In Section 2.4, we choose a state-of-the-art approach to serve as a starting point for our work and provide a justification for this choice. Section 2.5 then addresses a key research gap by proposing a generalized system model for component-based systems that captures that both software and hardware may be organized in a hierarchical fashion to address the complexity of complex systems. Section 0 then explains how to extend our selected baseline approach to consider such hierarchical system models, while Section 2.7 concludes with an evaluation of the extended synthesis approach, both using a simple synthetic case study and a more complex industrial case.

2.1 Introduction to System Synthesis

Platform-based design [3] allows new system variants to be composed from a repository of reusable components that may independently evolve over the lifetime of the system. Leveraging the principle of reuse has enabled the industry to reduce the time required to specify new system variants. However, manually selecting the software components to integrate—and how to deploy them on available hardware—still requires significant effort and expertise from increasingly scarce engineers. This challenge can be addressed through *automatic system synthesis*, which aims to satisfy both functional and non-functional requirements. System synthesis therefore supports a further *shift left* and optimization of the design phase, extending platform-based design.

The key idea of synthesis of component compositions and deployments is to leverage *models of components* that describe the *services* they provide to the system. These services are specified along with relevant properties, such as the quality level with which the service is provided, e.g. in terms of frames per second or image quality, and the amount of time and resources, such as CPU and memory, required for the component to perform its function. The models also specify the constraints under which they can deliver their services, which may include requirements on other services with particular quality levels being provided to them by other components.

Given a high-level specification of the required functionality, properties, and timing constraints for a new system variant, it is possible to express the synthesis of component compositions and deployments as an optimization problem that tries to find a set of components to integrate and a deployment on the available hardware (also described in models) that satisfy all specified requirements. Since there may be many possible solutions to such a problem, an objective function is also provided to determine which of the possible solutions is considered best, e.g., the one that maximizes performance, minimizes cost by reducing hardware over-dimensioning, or a combination thereof. This is an example of model-based design space

exploration, a topic that requires further attention according to a recent survey among industry practitioners [6].

System synthesis has several potential benefits. First of all, it provides automation, which helps increase engineering productivity. Secondly, it democratizes system integration by embedding much knowledge in the component models, reducing the dependency on scarce domain experts. For example, the use of models may aid system engineers in exploration of different configurations on actual software with only limited support from software engineering, increasing their independence and productivity [4], [5], [6]. Lastly, the element of automatic design optimization may help increase the quality or performance of the system, while reducing cost. These benefits are particularly important for complex systems with many reusable components that can be combined in many different ways to realize a particular functionality. A potential drawback of the approach is that it requires up-front modelling effort to create the required component models, a challenge shared by all model-based methodologies [7]. The quality of the result will also be directly dependent on the completeness (are all constraints considered?) and accuracy (is the timing behavior of the component accurately determined?) of the component models.

2.2 Requirements and Scope

This section describes the scope and a set of requirements that our industrial context imposes on our methodology.

The synthesis of software component compositions can be done at design time to configure new product variants or evolve existing ones. The case of evolution can be seen as a special case that involves fixing parts of the system that should be kept intact using component constraints that restrict the design space. For design-time synthesis, it is reasonable to expect a high-quality configuration in less than a day, a substantial improvement over the current practice of doing this manually. Longer synthesis times are expected to make the approach impractical, especially if several iterations are required in light of frequently evolving requirements.

System synthesis may also add value at run time in an operational system. In this case, the system can synthesize a new configuration when a piece of hardware unexpectedly fails in a deployed system, ensuring minimal performance degradation. Here, synthesis cannot leverage all reusable components in the repository but is limited to the subset of components available in the operational system. The priority in this case is to find a new configuration that satisfies constraints very quickly, e.g., in less than one second, to keep the system operational. The TechFlex 2025 project focuses on solving the design-time synthesis first and considers reducing synthesis time to support run-time use as future work.

The system model used in the methodology needs to reflect that complex systems are built hierarchically through decomposition into subsystems that may in turn be recursively built from subsystems or a set of components implementing a given functionality. This structure allows properties and constraints to be defined at either the subsystem or the component level, reflecting how systems are actually engineered. Components may provide or require multiple services on multiple interfaces, which means that the component model must be able to handle components with multiple inputs and outputs. Furthermore, large industrial systems may need the services provided by a subsystem or a component in multiple places, either to realize different functionalities or to duplicate a particular functionality, e.g. to simultaneously

decode multiple video streams. This raises interesting questions with respect to the synthesis process. Can components be shared between subsystems, or are instances per subsystem required? These questions need to be addressed by the proposed system synthesis methodology.

The result of the synthesis not only has to satisfy functional requirements, but also non-functional requirements, e.g., related to system performance. In this work, we consider a system with timing constraints specified as *system flows*, i.e., end-to-end latencies for sequences or graphs of interacting components. An example of such a system flow, modelled as a UML sequence diagram, is shown in Figure 2. The figure shows that the maximum time between a *SubmitOrderRequest* arriving at the *PlannerService* and the corresponding *DeliveryUpdateNotification* arriving back at the *PlannerService* should not exceed 250 ms.

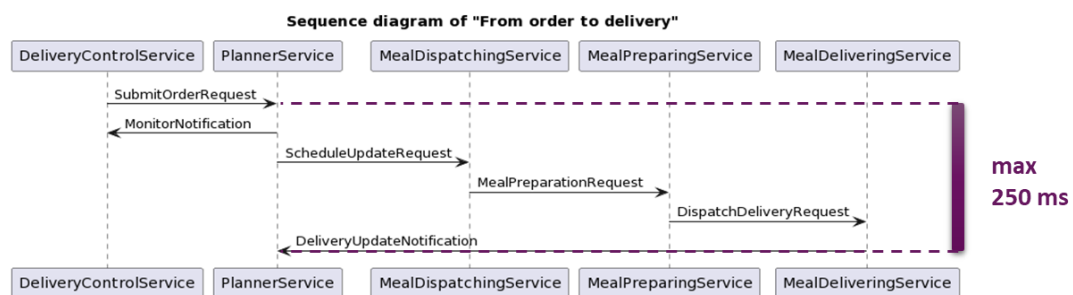


Figure 2: An example of a system flow, modelled as a UML Sequence diagram.

Complex software systems may be deployed on hundreds of compute nodes with heterogeneous configurations, e.g. in terms of number of cores, memory size, and availability of accelerators, like GPUs. These nodes may also be organized hierarchically, e.g. in different racks connected by different networks. The timing behavior of the system depends on both the selected components, as well as on how they are deployed over the compute nodes. To determine whether a particular deployment is valid, it must be possible to specify the available hardware resources, i.e. the compute nodes with their cores, memory, and potential accelerators, as well as the resources needed by components to perform their function. It must also be possible to predict the latency of components sharing resources, such as CPU cores and memory, of a compute node. This is the topic of RQ3 of the project of which the results are described in [8].

Lastly, the developed methodology must be compatible with existing engineering processes and timescales to be useful in industrial practice. The method must be supported by tooling that performs the service composition, as well as the deployment on available, heterogeneous hardware.

2.3 State-of-the-Art

A brief study of the academic literature revealed related work in several fields, e.g. component-based software synthesis, service-oriented architecture synthesis, and formal methods in systems synthesis. Much of the academic research has been conducted in this area during the first two decades of this millennium, often building on foundations of synthesis from the 60's through 80's.

In this section, we will review a few selected works and position them with respect to these dimensions.

- How components are modelled, e.g. with a single input and output or with multiple inputs and outputs
- Whether or not they consider quality-of-service (QoS), such as latency or throughput constraints
- Whether they model any type of system resources, e.g. CPU and memory, that are required by selected components to realize their functionality
- Which kind of optimization method is used, e.g. specialized heuristic methods [9], [10], general meta-heuristics [11], or mathematical programming [12], [13].
- How they were evaluated, e.g. not evaluated, evaluated with synthetic workloads [10], [13] or evaluated with a small example [12].
- Whether the approach is supported by an available tool [11], [12]

The work in [9] discusses the evolution of automated component-based software development tools, a story going from methods based on structural synthesis of programs (SSP) [14] to behavioral types. It discusses a number of tools, the older CoCoVila [15] and SoCoSys [16], as well as the newer BSynth. Both SoCoSys and CoCoVila build on SSP, but CoCoVila has a more general component model that considers a set of input ports, a set of output ports, constraints and credentials. The latter are used to capture non-functional properties, such as QoS. Both approaches use specialized heuristic algorithms to perform the software composition. Another similarity is that both SoCoSys and CoCoVila assume *completeness* of the repository, i.e. that there are available components in the repository that allows the puzzle to be solved. BSynth addresses this limitation by creating a set of *virtual components* that cover all combinations of dangling inputs of real components to dangling outputs. A heuristic search algorithm then tries to: 1) find a complete solution, 2) minimize the number of virtual components, and 3) minimize the total number of components. This work is related to ours, but considers automated synthesis of components, rather than of component compositions. Neither SoCoSys nor BSynth are available online. CoCoVila [17] is available on GitHub, but has not been maintained for six years.

A component language and toolset for Quality and Resource Management (QRM) is presented in [12]. It builds on mathematical component model elaborated in [18]. A domain-specific language (DSL), defined in Xtext [19], is used to specify components with multiple inputs and outputs, required/provided resource budget, and parameters. The DSL complements other component models in the sense that it only specifies QRM aspects of components and needs extensions to go beyond this. For example, it could be combined with component languages focusing on structural and behavioral aspects, such as ComMA [20], [21]. The QRM approach supports specification of different design alternatives, e.g. different software or hardware implementations. Model-to-text generators automatically generate a constraint problem formulations for the Z3 Satisfiability Modulo Theories (SMT) solver [22]. Solving the generated problem allows different alternatives to be explored and feasible solutions visualized as a Pareto front [23]. A tool for creating QRM models is available for online usage [24]. A framework for resource management and deployment of QRM components in resource-constrained environments is presented in [25]. Overall, this approach is strongly related to the research presented in this work. The main difference is that it does not allow hierarchical specification and synthesis of systems, from subsystems down to individual components, which is typical in engineering of high-tech equipment.

A service composition problem in the context of service-oriented architectures is addressed in [13]. The challenge is to compose services from a repository of components that best satisfy a targeted orchestration model in terms of non-functional qualities, such as QoS, reliability,

and cost. Orchestration models are similar to workflows and are modelled as a graph with tasks and gateways, where tasks have a single input and output, and gateways are orchestration components like AND/OR/XOR and split/join. An example of an orchestration model from [13] is shown in Figure 3.

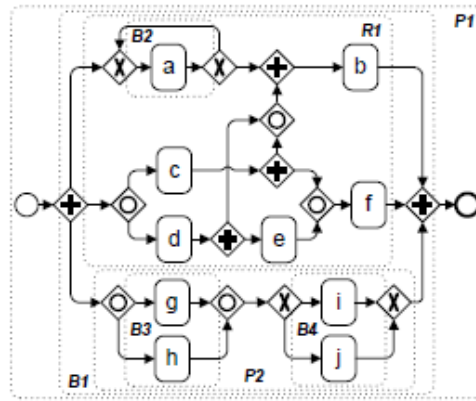


Figure 3: An example orchestration model [13].

They also provide aggregation functions that mathematically describe how different non-functional qualities, such as execution time, reliability, throughput, and cost, are affected by the orchestration components. This allows an analysis to be made for the different paths through the models. The problem of selecting the set of component implementations for the different services in the orchestration model is formulated as a non-linear program, which is then linearized and solved as an Integer Linear Program (ILP) using IBM ILOG CPLEX Optimization Studio [26]. This result is compared to a proposed heuristic algorithm, which allows results with slightly lower quality to be computed in shorter time, for a large number of synthetic problem instances. In both cases, the synthesis of service compositions is finished in less than 100 ms for up to 40 candidate services, each with 10 different possible implementations.

This research has both important similarities and differences with our work. A key difference is that it only considers components with a single input and output, and does not model required system resources, such as CPU and memory, required to compute deployments. An important similarity with this work is the way it captures non-functional qualities end-to-end. We expect the orchestration components and aggregation functions to be important ingredients when considering end-to-end latency requirements for our system flows.

Another work addressing service composition is presented in [10]. This work focuses on run-time service composition and particularly considers the time required to perform the composition in a dynamic environment, where there are many requests for compositions, as well as frequent changes in availability of services or in their QoS properties. A heuristic algorithm is proposed to compose systems in a forward-chaining way, while considering the required QoS. The approach is evaluated with for a set of synthetic cases with variable complexity and shows that composition can be done in a few hundred milliseconds with up to 1000 services. There is no mention of an available implementation of the algorithms in a tool. Similarly to our work, this research addresses a service composition problem, but contrary to our ambitions, it focuses on the efficiency of run-time composition in a dynamic environment.

ADMORPH [27] was a European project that focused on specification, design, analysis, and runtime deployment of adaptive cyber-physical systems. As a part of this project, an approach to run-time synthesis of service compositions was proposed [11]. In this approach, components are modelled according to the OSGi capability-requirement model [28]. Components provide and require services, where provided services may be specialized by attributes that describe its properties. For example, an image processing service may have attributes like execution time, maximum resolution, quality level of the processed result, and amount of hardware resources, e.g. memory and CPU, that are required to perform the computation. Similarly, required services can have filters that describe what properties are expected from the provided service. When matching provided and required services, the filters and attributes are compared to determine whether a match can be made. A similarity with QRM [12] is that components, as well as available hardware, are specified in DSLs made with Xtext [19].

The design space of possible solutions is modelled as a Directed Acyclic Graph (DAG). At the top, the required system services are specified. Then, for any required service, components providing the service are recursively connected to the graph. Any valid design point in this space is a complete subset of this DAG, i.e. where all required services for all the components are matched to a provided service of another component. Note that by also including hardware as a required service and compute nodes as hardware service providers, the deployment of software to hardware is encoded in the design point and optimized together with the component composition. When deploying a component to a particular compute node, it consumes a specified number of compute cores, as well as memory, from the hardware node. This captures that compute nodes have finite resources and can account for performance aspects expressed in terms of resource utilization.

The ADMORPH optimization of the service composition, as well as deployment on the available hardware, is performed using a meta-heuristic Genetic Algorithm [11], [29] implemented using Distributed Evolutionary Algorithms in Python (DEAP) [30]. The algorithm evolves a population of random possible solutions during several iterations to improve their fitness. There are several objective functions that define different optimization criteria, e.g. minimizing hardware cost, average load, or maximum load. The population size and iterations are programmable hyper-parameters, providing a computation/accuracy trade-off. The approach is evaluated using synthetic problem instances and results show that good results are often achieved in a few seconds. The approach is supported by a prototype tool, developed by Thales [11].

The various available methods make clear that a synthesis approach is feasible. The key challenge will be to scale a method to an industrial size complex, hierarchical, hardware constrained environment and provide sufficient tooling.

2.4 Choosing a Starting Point

After describing our requirements and scope in Section 2.2 and reviewing the state-of-the-art in Section 2.3, we now proceed choosing a starting point for the work in TechFlex.

The QRM [12] and ADMORPH [11] approaches are the best matches for the stated requirements as they can model components with multiple inputs and outputs, resource requirements, as well as specify QoS aspects like latencies and quality levels. Both approaches specify component models using DSLs made using Xtext, fitting the context of our work which already makes extensive use of this technology [3]. QRM has the benefit of

having a solid mathematical foundation, but ADMORPH is already known within (parts of) the target organization, which simplifies transfer and adoption of research results. ADMORPH was originally developed to also cover run-time synthesis. Since the genetic algorithm used in the ADMORPH approach provides a programmable trade-off between latency and accuracy, the need for high-quality solutions can be traded for quick solutions that satisfy requirements, allowing the system to stay operational if a problem is detected with a software component, or if a compute node unexpectedly becomes unavailable.

While the QRM and ADMORPH approaches fulfill many of our requirements, neither approach, nor any of the other reviewed approaches, address the requirement to be able to model and synthesize systems that are hierarchically decomposed into subsystems. Instead, all reviewed approaches consider systems comprising a single level with components, without any further structure or hierarchy. Both approaches can describe performance aspects of components, such as the latency required to perform its computation. However, neither approach prescribes how to obtain this latency for different deployments with different co-location scenarios. Lastly, none of the approaches have been evaluated on an industrial case study of representative size. ADMORPH has been evaluated with synthetic case studies of various sizes to assess its performance and scalability [11], while QRM has primarily been demonstrated using smaller illustrative case studies [12].

Based on the above comparison, we choose ADMORPH as the starting point for the TechFlex approach to system synthesis and use it as a vehicle to explore RQ1.

2.5 Modelling Hierarchical Systems

Having selected the ADMORPH approach as the starting point for the TechFlex approach to systems synthesis, we start by addressing one of its key limitations, namely that it cannot model and synthesize complex industrial systems that are hierarchically decomposed. To address this limitation, we first propose new generalized meta-models that can describe (software) systems (Section 2.5.1) and hardware configurations (Section 2.5.2) that are hierarchically structured.

2.5.1 Hierarchical Software System Modelling

To enable synthesis of software systems that are hierarchically decomposed, we propose a very generic recursive system meta-model, shown in Figure 4. In this meta-model, a *(sub)system* exposes an interface with one or more provided/required *services* to the external world. Each (sub)system in this hierarchy comprises one or more *parts*. Parts are logical entities, templates that expose an interface that needs to be physically implemented. Each part can be implemented by either another (sub)system or a *component*, as long as it complies with the (interface) specifications (provided and required services) of the part. (sub)systems, parts and components are jointly indicated as *nodes* in this model. The (sub)systems are aggregators of functionality, while the actual implementation is located in the components. The top-level system is fully self-contained and does not expose or require any external services.

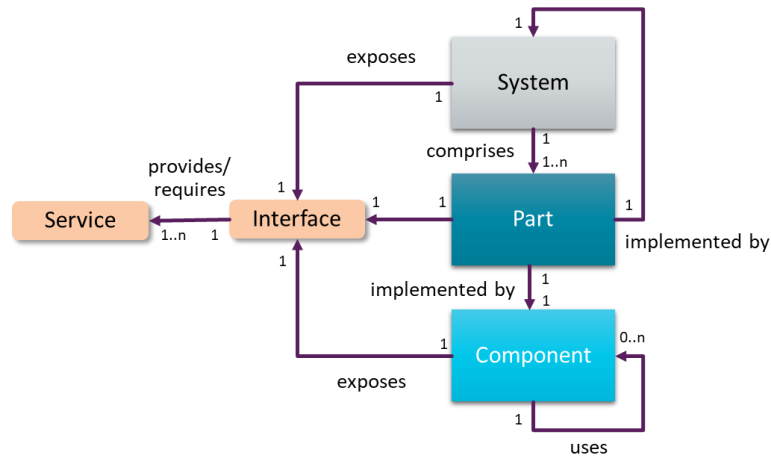


Figure 4: System meta-model that describes the hierarchical structure of a system and its implementation.

Figure 5 illustrates a subsystem exposing an interface that requires a Service A and provides Services X and Y. This subsystem comprises three parts, P1, P2, and P3, that perfectly match the interface of the subsystem when combined. Each part may be implemented by a component or another subsystem (not shown). The proposed recursive model hence supports decomposition with an arbitrary depth. The figure also illustrates that timing requirements can be specified as properties at the level of subsystems. This means that they can be specified at any level of decomposition, without knowing how the subsystem will actually be implemented. This provides a way to address the type of timing requirements on system flows previously discussed in Section 2.2.

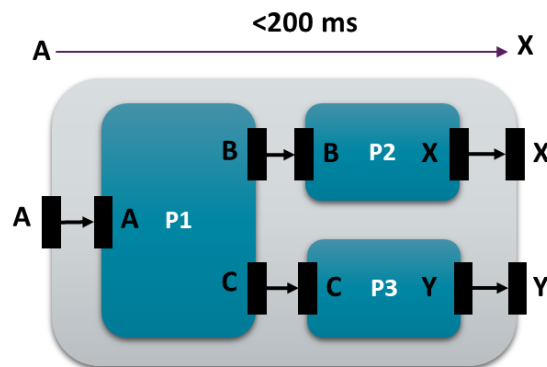


Figure 5: A subsystem implemented with three parts.

Figure 6 illustrates the case where a component is implementing a part, in this case Part 1 from Figure 5. Note that the component (Comp. A) fully matches the interface of the part, but may recursively use other components (Comp. B and Comp. C) to provide its functionality. Those supporting components may not require or provide any services on its external interface.

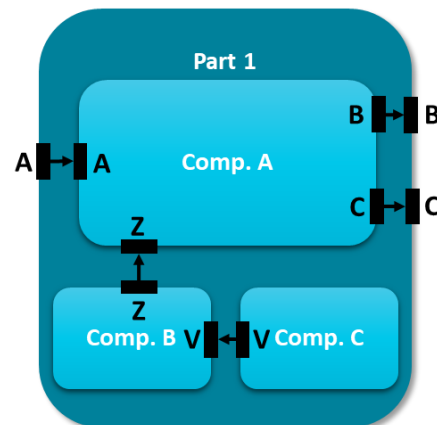


Figure 6: A part that is implemented by a component, which in turn uses other components.

Figure 7 shows an example of a hypothetical System A described using the meta-model. System A contains three distinct parts. Part 1 and Part 3 are further decomposed into subsystems, each that are in turn realized by a fixed number of parts. In contrast, Part 2 has been directly implemented by a single component (Comp. D) that matches its provided and required services and does not use other components. In the end, all functionality in the system is implemented by components.

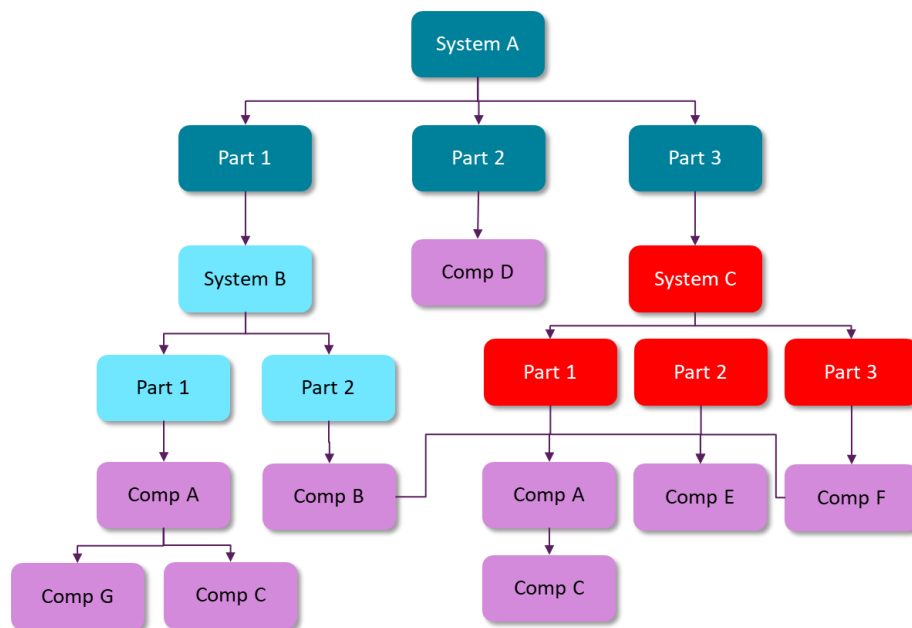


Figure 7: Example of an hierarchically decomposed system.

The proposed generalized hierarchical system model has been implemented as domain-specific language in Xtext for easy use in our industrial context. This language replaces our earlier version of the 'Module DSL', described in the TechFlex 2024 report [3], which could only represent a system as a flat set of integrated components. It allows synthesis to operate on models that reflect the abstraction levels used in practice, making the resulting methodology more transferable to industrial engineering workflows.

2.5.2 Hierarchical Hardware Modelling

Having discussed how to model a (software) system and its hierarchical decomposition into subsystems, we proceed by describing the model of the hardware onto which the components will ultimately be deployed. The hardware model in the ADMORPH approach describes a set compute nodes connected by a network. Both compute nodes and networks have properties that can describe key aspects relevant for resource management, such as the number of cores, the amount of memory, or the provided bandwidth. While this model provides a good starting point for TechFlex, it does not quite capture that large applications in an industrial environment may use very extensive hardware configurations comprising hundreds of compute nodes and network components. To adequately address this complexity, we propose a generalized hardware meta-model, shown in Figure 8, that is able to describe the organization of such hardware configurations in a hierarchical structure.

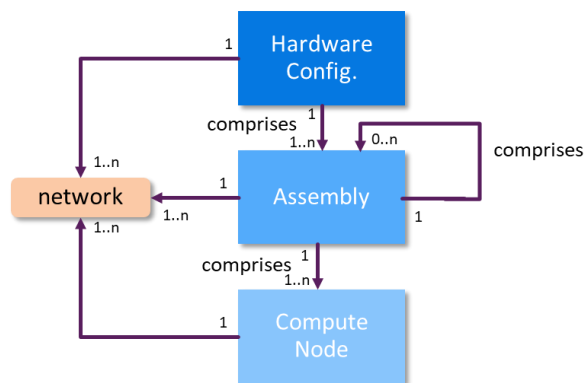


Figure 8: Hardware Configuration Model

A hardware (configuration) consists of *assemblies*, representing a set of compute nodes, typically packed in a rack, serving a dedicated purpose for the application (e.g. image processing). To allow the description of very large hardware configurations, the assembly is a recursive element that can contain zero or more (sub)assemblies. This may be used to describe a hierarchical structure where an a processing cabinet comprises a number of racks, each containing several compute nodes. Just like in the system model, proposed in Section 2.5.1, the leaves of the model, the compute nodes, contain the actual computing functionality. Assemblies and hardware are mainly structural elements. A compute node is typically a single computing board with one or more CPU's, each containing multiple cores, GPU's, and memory, as well as one or more network connections. These resources are described as properties of the respective node.

An instance of the proposed hardware meta-model with two assemblies is shown in Figure 9. Each assembly has a network connecting its nodes and a third network is connecting the two assemblies. In total, this hardware configuration has five compute nodes.

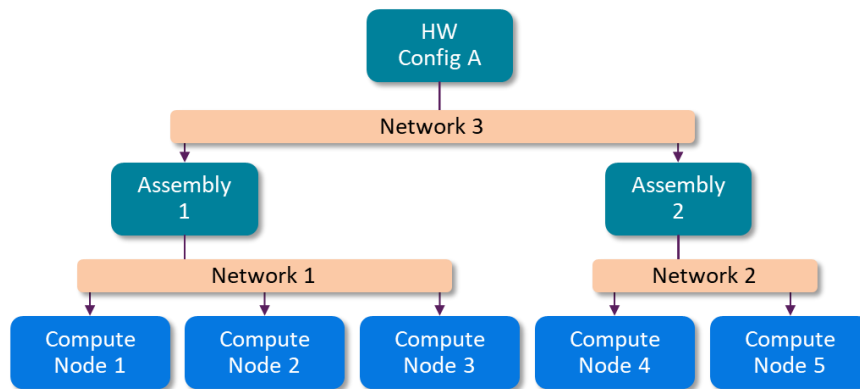


Figure 9: An example of a hardware configuration with two assemblies.

The synthesis process uses an instance of this hardware meta-model and maps all components of the solution (an instance of the system model presented in Section 2.5.1) onto compute nodes. Note that multiple components may be mapped to a single compute node, taking into account the available resources (memory, cores) on the node, allowing the synthesis process to find deployment alternatives under realistic constraints.

The hierarchical hardware models used in this work are currently implemented using a simple text-based representation. In the future, it could be implemented more elegantly as a DSL, similarly to the system meta-model. However, this was outside the scope of this work.

2.6 Synthesis of Hierarchical Systems

Having generalized the system and hardware meta-models to include hierarchical decomposition, this section discusses how to extend the ADMORPH approach, selected as the starting point for this work, to support that generalization. First, in Section 2.6.1, we discuss how to integrate the hierarchical concepts of parts and subsystems into the ADMORPH synthesis process with limited changes. Next, in Section 2.6.2, we describe how to create the design space of valid synthesized solutions, given the architectural principles of the system being defined. Lastly, we provide an overview of the synthesis process and how it is implemented in scripts in Section 2.6.3.

2.6.1 Supporting a Hierarchical System Model

As previously mentioned in Section 2.3, the ADMORPH approach works by matching requested and provided services. It recursively creates a Directed Acyclic Graph (DAG) with all possible components that can provide the required services, representing the design space of possible implementations. In this work, we extend this DAG to consider our generalized system model from Section 2.5.1 by introducing Systems and Parts that may be implemented by further decomposition into subsystems, in addition to being directly implemented by components. We want to make this work with limited modifications of the original approach and tool by representing the hierarchical structure in a form that can still be processed by ADMORPH's existing service-matching logic, rather than replacing that logic with a new synthesis engine.

Therefore, we use the following three steps:

1. According to the generalized system model, (sub)systems comprise a fixed number of parts. By adding a unique virtual required service to each (sub)system and a corresponding virtual provided service to a related part, the synthesis algorithm will always connect a system to its constituent parts, as they will be the only providers of those virtual services.
2. For each part, identify based on its interface (specification of provided/required services), which subsystems and components are candidates for its implementation. Add a unique virtual required service to the part and a corresponding provided service to each of the implementation candidates. This ensures that both components and subsystems with matching interfaces are considered by the synthesis algorithm.
3. All requested and provided non-virtual services of systems and parts are removed, as they are no longer needed due to the previous two steps. The virtual services will ensure that systems and parts are properly matched.

These three steps enable the existing ADMORPH synthesis tool to synthesize hierarchically decomposed systems based on our model, by recognizing that systems comprise a fixed number of Parts and will correctly identify both subsystems and components as implementation candidates, assuming their interfaces match. As a result, the ADMORPH-based synthesis can be applied to hierarchically decomposed system architectures of the kind found in industrial practice.

2.6.2 Creating the Design Space

A large system may require multiple instances of the same (sub)system. For example, consider a large image processing system that handles multiple video streams, each being processed by its own instance of a processing subsystem. To synthesize such a system, we need to encode reuse principles on when to share components between subsystems or create separate instances into the design space explored by synthesis. In addition, we need a practical solution to allow the synthesis process to distinguish between different instances when connecting the required and provided services of the components.

The reuse principles mentioned will be based on the system under consideration and its architectural principles. To support this definition, we need to introduce a few additional concepts. The *scope* of a node (system, part, or component) is defined as the path going from the root of the graph to the particular node, expressed in the form of a directory structure. Each system and part in the graph is represented as a next-level directory in this view. This is illustrated in Figure 10, which shows an example of System A, previously introduced in Figure 7, and its mapping to the compute nodes in the hardware configuration from Figure 9. In this figure, the five nodes with encircled numbers in Figure 10 have the following scopes:

1. The scope of System B is: /SystemA/Part1
2. The scope of Component A is: /SystemA/Part1/SystemB/Part1
3. The scope of Component A is: /SystemA/Part3/SystemC/Part1
4. The scope of Component C is: /SystemA/Part1/SystemB/Part1
5. The scope of Component E is: /SystemA/Part3/SystemC/Part2

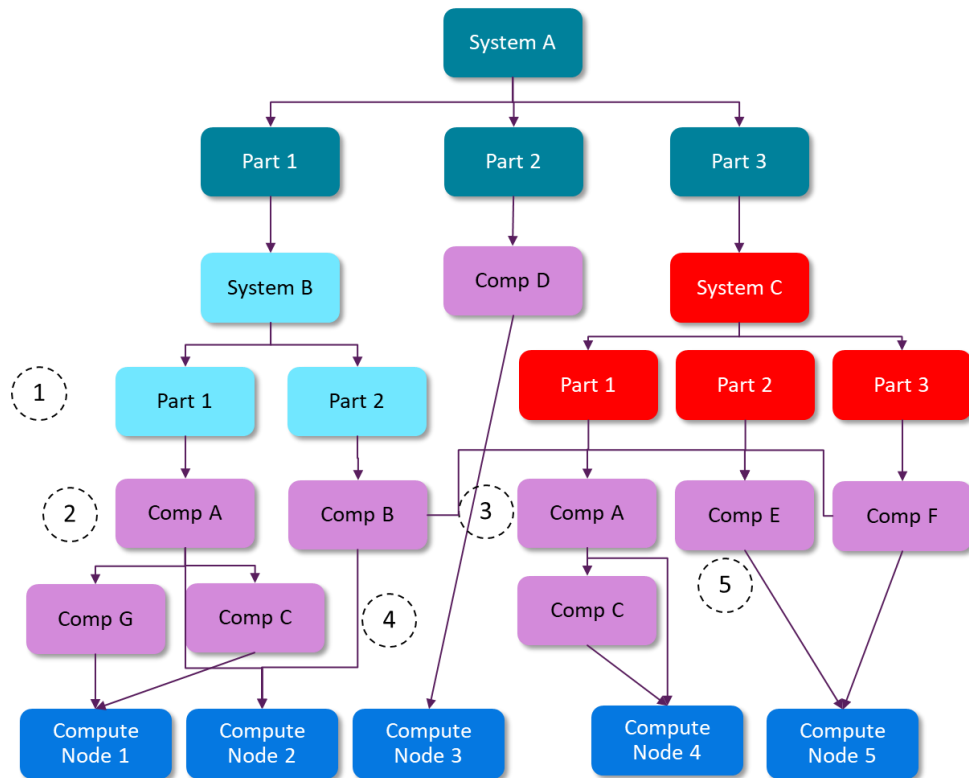


Figure 10: An example of a synthesized hierarchical system and its deployment on hardware nodes.

The definition of scope allows us to introduce the concept of *locality*. Local to a particular subsystem or part refers to nodes downstream from its scope. For example, local to (sub)system B means everything that has as scope 'System B/*'. The opposite of being local is referred to as being *external* to the scope. Finally, we define the *path length* between two nodes (from required to provided service) as the number of 'directory steps' to be made traversing the graph from source to destination. The path length between Component A (Example 2 in Figure 10) and Component E (Example 5) is 7 steps.

For TechFlex, the following strategy is used for connecting provided and required services in our industrial case study. Note that the method can be easily adapted to support other strategies. Services that are required by the implementation of the parts in a (sub)system (i.e. not visible on the interface of the (sub)system) are implemented locally. This means that new components will eventually be instantiated to facilitate the implementation. As an example, the required service Z of Component A in Figure 6 does not appear on the external interface of its part, so a provider must be instantiated locally, resulting in the instantiation of Comp. B (or possibly another implementation candidate, if there are multiple). In contrast, Service A of Component A is required on the interface of the part, which means that the synthesis algorithm will look for providers of this service outside the scope of the part. In case there are several components providing the requested service externally, the one with the shortest path is chosen to provide the required functionality as closely as possible in terms of the system architecture.

Next, we discuss how this is implemented in the synthesis tool. This is done by first creating the DAG representing the complete design space, starting at the top-level system. For each required service that should be implemented locally, i.e. the service is not visible on the external interface of a part, all possible subsystems and components providing the service

are connected. Note that the resulting DAG is only finite when there are no cyclic dependencies in the repository of reusable components used for synthesis. To include this design space in the synthesis process, the ADMORPH approach creates a dictionary that lists potential providers for all required services for all nodes. This dictionary is used to generate candidates for the genetic algorithm. To cover the distinction between services that are local and external to a particular scope, this dictionary has been extended such that it can consider the length of the path to the providing node. This enables synthesis to take place in two steps. First, the maximum path length is set to 4, effectively enforcing that only provided services from the local scope are in the dictionary. Assuming the repository of reusable subsystems and components contains providers for all required services, synthesis will find an implementation for all local required services. A second step considers the remaining, external, services. For this, a new dictionary is generated that is restricted to the components that have already been synthesized in the first step to ensure that existing components are shared rather than duplicated, as defined by our architectural principles. Here, the component with the shortest path length is always selected.

In principle, any component can be deployed to any compute node, unless it has specified that it needs a particular resource, such as a GPU, to execute. This can be expressed as a property that is used by the tool to filter the design space. Similarly, it is possible to indicate the affinity of a component, e.g. that it prefers to be deployed on a particular compute node, or in a particular assembly. This gives system designers some control over software deployment and restricts the design space. The possible deployment does not initially consider resource utilization, i.e. availability of cores and memory, since it is not yet clear which of the possible deployments will actually happen. Instead, this is considered as a constraint when candidate design points are given to the genetic algorithm.

2.6.3 Overview of the Synthesis Process

After explaining how the ADMORPH synthesis approach has been extended to support a hierarchical system model (Section 2.6.1) and creating a design space of valid solutions for the Genetic Algorithm (Section 2.6.2), we now provide a brief overview the implementation in the synthesis tool. An overview of the complete five-step synthesis process used by the tool is shown in Figure 11.

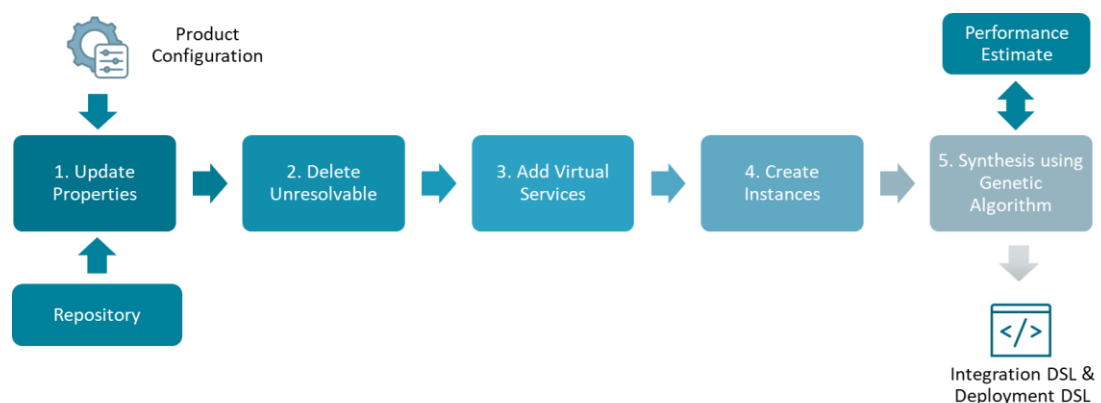


Figure 11: Overview of the resolver.

A key input of the process is the requested product configuration, which is a top-level system consisting of a number of parts whose implementation should be synthesized. For example,

this could be a complete system comprising a data acquisition part and a data processing part. This configuration can also describe required properties for the implementation of each part, e.g. a desired quality level.

We will now walk through the five steps of the synthesis process in more detail. This process leads us from a project repository consisting of a description of all hardware and software nodes towards an optimized solution expressed in the form of an integration and deployment description :

1. The total repository of reusable nodes (subsystems, parts, components) can be updated with the properties set in the product configuration. For example, a product configuration may state that it wants a particular part of the system to be implemented using a simulator rather than real components that may depend on hardware that is not yet available. In this case, this will be used as a filter during synthesis to limit the provided implementations to simulators. Encapsulating this in the product configuration rather than requiring this as a property further down the hierarchy reduces complexity by avoiding unnecessary duplication of nodes, e.g. one part that requires an implementing subsystem to be a simulator and one that does not.
2. A validation step makes sure there are no unresolvable elements in the repository. If there are, they are removed from the repository and a warning is issued to the user.
3. Virtual services are added to allow the hierarchical system structure to be considered during synthesis (Section 2.6.1). First, it is established what systems or components could be selected to implement each part and the virtual services are added to systems, parts and components, replacing the original services on the interfaces.
4. The design space of possible valid solutions to be considered by the genetic algorithm is created (Section 2.6.2)
5. Synthesis using the genetic algorithm. For each considered design point, the performance estimator estimates relevant performance metrics. The performance estimator could be based on any method that is can predict performance for a given set of components and their deployment, e.g. [8], [31].

There are two main outputs from the synthesis process:

1. A description of all component instances that must be generated for the solution. This is described in the so-called Integration DSL, introduced in [3].
2. A description of the deployment of the synthesized solution, with for each component a directive for its deployment to a specific node. This is described in the so-called Deployment DSL, introduced in [3].

The synthesis process has been fully implemented in Python 3 The total synthesis suite consists of five scripts that:

1. Collect all component, system, specification and hardware description files from a repository.
2. Convert all these descriptions into a format suitable for the synthesis tool and check their internal consistency and completeness, e.g. are there referenced

specifications missing? Do the parts of a system together offer the services as specified in the system?

3. Prepares the system structure and design space, as described in Sections 2.6.1 and 2.6.2, respectively. This corresponds to executing the steps from 'Update properties' until 'Create instances' in Figure 11.
4. Synthesize, starting from a top-level product description, leading to an optimized solution.
5. Output results of the synthesis in various forms.

2.7 Evaluation

Having extended the ADMORPH synthesis approach to handle increasingly complex systems by introducing hierarchical system and hardware models, we proceed with validating and evaluation our extensions. This is done in a two stage process:

1. First a small synthetic case was used for a basic functional verification of the developed scripts and the extension to cover hierarchical system models.
2. A larger industrial case was used to validate the synthesis process and its performance at relevant scale.

All functional validation of the implementation of the method was done by an expert, manually comparing synthesis results to expectations.

2.7.1 Synthetic Evaluation

A small synthetic case was used for initial verification of the synthesis process, depicted in Figure 12. This case considers a small system with eight components. It has four control components (bottom left) that perform planning, dispatching, preparation, and delivery. Added to these control services, three very data intensive components are available (bottom right): one generation service, generating a tunable amount of data, a forwarding service, copying the data forward to create memory load, and finally a data reduction service that consumes the data and processes it with a Fast Fourier Transform to generate (CPU) load and reduce data. Finally, a delivery control service takes care of the 'user' side of the system by sending requests to the other services. This case simulates a combined control and processing system with substantial CPU load. To test the hierarchical element of the resolving, 6 (sub)systems were created. For some of these components, variants were included with different quality levels (attributes) to generate a solution space for the verification of the synthesis process. In addition, a hardware configuration of 3 compute nodes was created for the deployment of the case. Despite the limited set of nodes in the case, there were already 96 potential component compositions that could be deployed in more than 6000 ways on the available hardware.

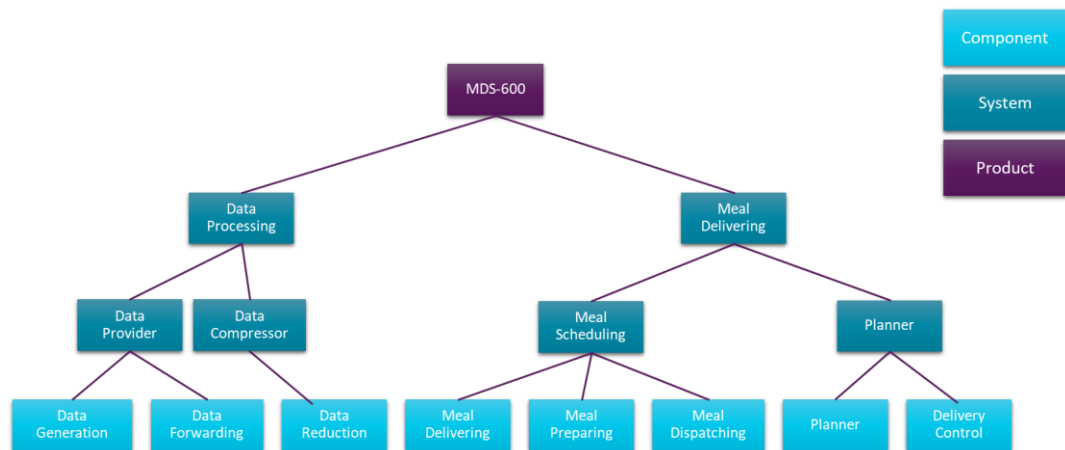


Figure 12: Structure of the synthetic test case.

This setup was used to verify the basic implementation of the synthesis process. This allowed verification of functional resolving, performance optimization and steering synthesis by setting quality attributes. The extension to cover hierarchical structure with parts and subsystems was also verified in this way. In terms of performance, the synthesis tool, configured to perform 20 iterations with the genetic algorithm, required less than 5 seconds to complete.

2.7.2 Evaluation with Industrial Case

A more complex (anonymized) industrial case was used to test whether the method remains workable under a level of structural and deployment complexity that better reflects the intended application domain. This case has the following:

- 70 interface specifications
- 57 component specifications
- 34 module specifications
- 154 compute nodes

Figure 13 shows a simplified version of this case, focusing on the top-level with the product the first levels of parts and subsystems.

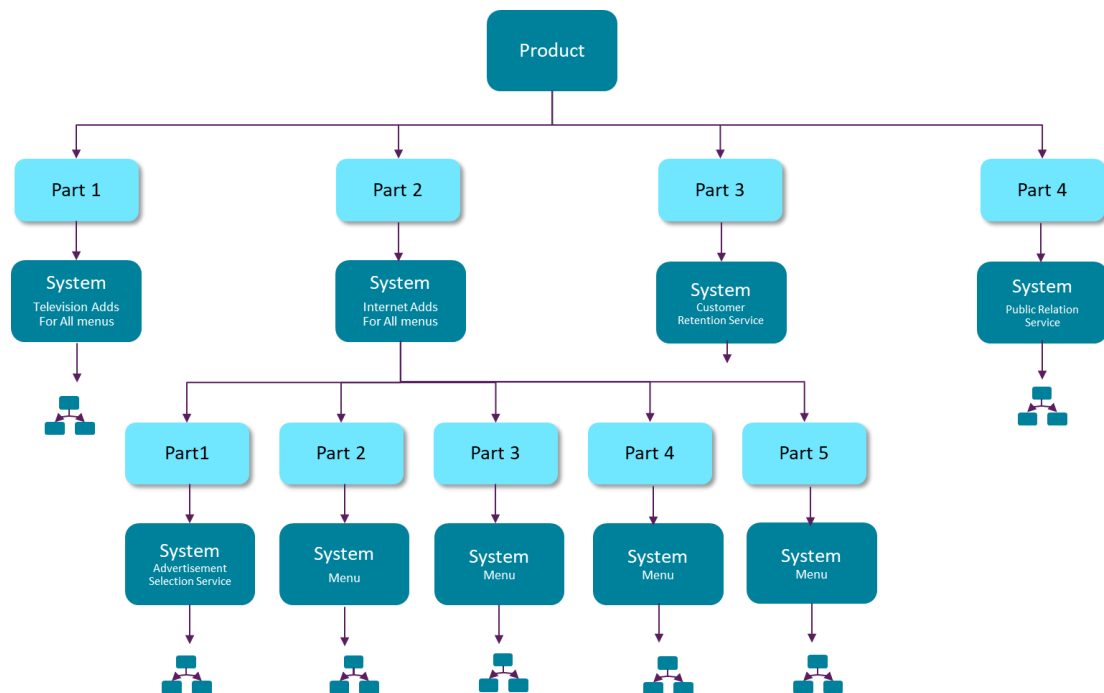


Figure 13: Parts of the solution of the industrial case

A single synthesized solution can have up to 400 instances of nodes included, depending on the optimization (objective function, attributes/filters) and the number of specified parts. The synthesis result for the industrial case was manually evaluated by experts and found to be correct.

As the main goal of the project was to research the algorithms and methods needed to perform the system synthesis for complex hierarchical systems, no extensive research on the metaheuristic optimization or performance aspect of the method has been done. The objective function used by the genetic algorithm was not varied during the experiments. The used objective function optimizes according to several performance parameters, maximum load per compute node, as well as ensuring sufficient usage of nodes to reduce the number of compute nodes.

A limited performance evaluation of the synthesis tool has been done as part of this work. All experiments have been executed on a standard laptop (AMD Ryzen 7 PRO 4750U, Radeon Graphics (8 cores, 1.70 GHz), 16.0 GB memory; Python 3; single-threaded application). Using this setup, synthesis of the industrial case takes about 3 minutes, clearly suggesting that it is fast enough for design-time use, even if requirements are frequently changing.

In this experiment, the genetic algorithm was configured with a population size of 15 and ran for 15 iterations. In addition, we varied these hyper-parameter values to estimate their influence on the results. The conclusion from this brief experiment was that the optimization typically converges after around 10 iterations, with only minimal improvements after 20 iterations. Population sizes of 10–30 were tested, and we concluded that a population size of 15 already gives good results. Increasing the number of iterations and the population size leads to increased computing time, as illustrated in Table 1. Configurations 1 and 5 were the fastest, but the genetic algorithm did not have time to converge in these cases, suggesting that it may have missed higher-quality solutions.

Table 1: Run-time of the synthesis for different hyper-parameter values.

Configuration	1	2	3	4	5	6	7	8
Population size	15	15	15	15	5	10	20	30
Iterations	5	10	15	20	15	15	15	15
Time (s)	121	162	232	274	114	143	271	596

3 Synthesis of Mock Components using ComMA

This section presents the results of our investigation into **RQ2**: *What defines a (minimal) set of component behavior and interface specifications needed to generate an executable system model that can be used to validate system flows and their performance in an early phase without the need for a complete model of all components?*

Section 3.1 presents the main motivations, goals and our strategy to tackle the challenge. Section 3.2 provides an overview of similar approaches and supporting tools, highlighting the main differences with our chosen approach. Section 3.3 provides an intuition for the syntax and semantics of the ComMA modeling language and a high-level description of the code generator implementing these semantics. Section 3.4 presents the demonstrator used to show the feasibility of such an approach.

3.1 Introduction

During the system design phases, key decisions are made by architects about the choice of hardware platforms (compute nodes) on which the software components are going to be deployed, and about the characteristics of the underlying middleware layer enabling interactions between them. These decisions have an impact on the timing behavior of the overall system. Verifying whether these design-time decisions satisfy the stated system requirements is usually only possible after the software modules have been developed and can be deployed on the actual physical setup for actual integration or acceptance testing. As a consequence, the validation of these critical design choices are postponed to much later in the system development process. When defects or non-compliance to requirements are detected at this stage, they have a huge impact on cost and time needed to resolve them, possibly leading to budget overruns and a failure to meet release deadlines. Whenever there are updates or upgrades to the hardware platform and/or middleware after a system has been qualified and released, testing must be performed again to validate that the system requirements are still met.

To prevent late detection of non-compliance with timing requirements, it is essential to develop *techniques* that allow architects to gain early design-time insight into whether (i) a network of software components will meet expected timing requirements on chosen hardware/software platforms and middleware, under expected communication patterns and load, and (ii) an update to hardware, middleware, or operating systems leads to non-compliance. For the latter, such a *technique* must enable effective and efficient troubleshooting to determine whether the detected non-compliance originates from the platform or from deployed software components.

Clearly, enabling such design-time analysis requires an unambiguous specification of the behavior of the individual software modules and their expected interactions with each other. To create such specification, systems modeling standards (e.g., UML and its variants such as SysML) are a good solution. They allow a designer to abstract away from implementation details and focus on the essential functional aspects. Such specifications can then be used

as an input to *code generation techniques* that generate mock software implementations conforming their specifications. Such mocks may then be deployed over target hardware and software platforms to simulate different deployment scenarios and check for conformance to timing requirements, without having to wait for the development of actual software modules. For cases where all the software components are available and an architect wishes to investigate the impact of an update or an upgrade to software or hardware platforms, the generated mocks can be very useful to distinguish whether non-conformance to timing requirements are caused by a change to the platform, or due to defects hiding in the software components themselves.

3.2 State-of-the-Art

Code generation [32] is an algorithmic technique used to improve productivity and quality of software development by automatically generating executable software code, or code fragments through predefined patterns, models, or specifications, which are in stark contrast to the traditional approach of hand-crafting code by software developers. Code generation approaches are commonly classified based on their inputs, which can be either (i) informal descriptions of requirements or design in natural language (e.g., BERT [33] and GPT [34] [35]), (ii) structured data that instead of free-format text provides pre-defined templates [36] to structure textual data (for e.g., JSON, XML, etc.), (iii) models capturing structure and behavior of the system, including its components, relationships, and flow of data between the components (e.g., UML [37]), (iv) images corresponding to screenshots, mockups, or hand-drawn images of user interfaces [38], or (v) existing code, to either optimize, refactor or repair by generating a new version of it, possibly in the same or different language [39] [40].

Although we see an explosion of interest in harnessing AI-based solutions (CodeLlama, DeepSeek, CodeT5, CodeBERT, GPTo4, etc.) to exploit natural language descriptions, generating complete and fully functional code without manual intervention is a challenge, largely determined by the complexity of the request, the clarity of the prompt, and the specific domain context of the required task. In this regard, the use of models as input for code generation presents the most promising solution. Among the many modeling languages out there, UML is the most popular, featuring class diagrams [41], sequence diagrams [42], state machines and activity diagrams [43]. Due to a lack of explicit semantics in the initial versions of UML, most of these approaches, including commercial tools such as Enterprise Architect¹ and IBM Rhapsody², have historically exploited programming languages to define behaviors and generate full-fledged code. To remedy this, an action semantics for UML was introduced called fUML with a textual front-end language ALF [44], for which code generation in C++ [45], among others have been proposed. Besides UML, code generation from modeling languages such as AADL [46], [47] have been proposed. Statecharts [48] are another popular modeling formalism for specifying discrete event systems supported by many tools (e.g., Itemis Create (formerly Yakindu) [49]) that support simulation and code generation in a variety of target languages.

When correctness of the generated code is critical, for instance in cyber-physical systems in medical, aerospace, and defense domains, the use of tools with a rigorous underlying formal semantics are essential, as well as the ability to verify specifications for the absence of deadlocks, livelocks, conformance to requirements, etc. In this regard, tools such as Popili/Cocotec³ and Dezyne⁴ have been developed and widely adopted [50] by industry.

¹ Sparx-Systems. Enterprise Architect. <https://www.sparxsystems.eu/>

² IBM Engineering Rhapsody [IBM Engineering Rhapsody](#)

³ Cocotec tool [Cocotec - Formal verification](#)

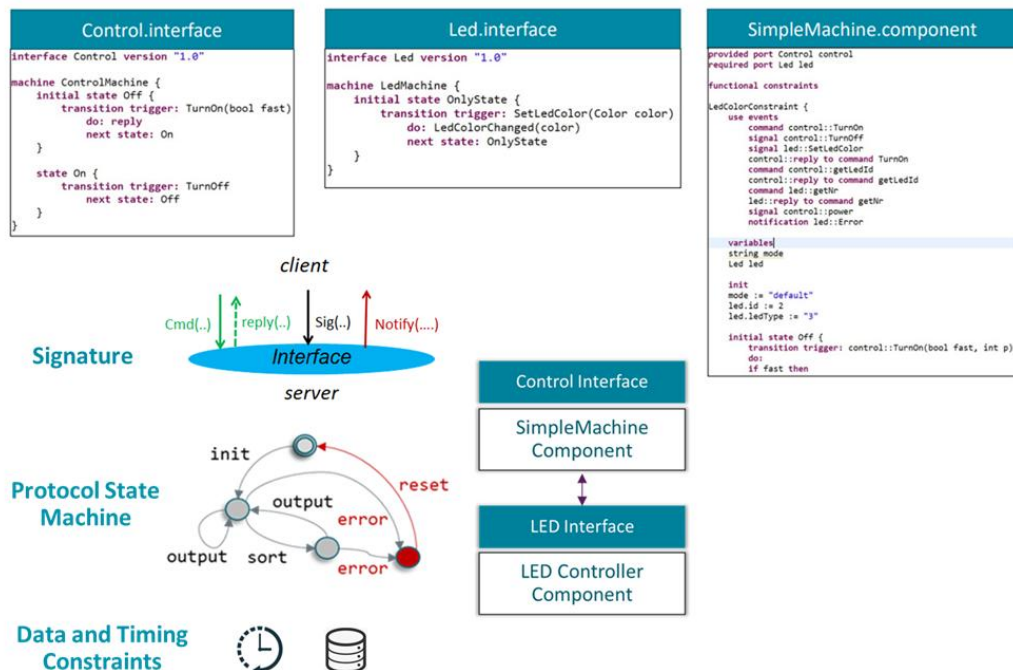
⁴ Dezyne tool verum.com

However, in the context of our research question this approach has two drawbacks: (i) they are closed-source based on a custom proprietary language, and (ii) they require a complete and deterministic specification of the behavior, since they are targeted towards generating full-fledged production quality software implementations.

The open-source ComMASuite⁵ tool with its family of domain-specific languages offers a promising alternative to specify interface and component behavior as state machines. ComMA was developed by TNO-ESI [53] in collaboration with Philips and Thales, based on the observation that poorly defined interfaces often lead to integration issues and costly rework. To address this concern, an interface modeling language was defined to capture the often-missing dynamic behavior aspects as protocol state machines. With such specifications, service providers can unambiguously define a contract (assumptions and expectations) toward their service consumers. Based on such contracts, run-time monitoring techniques can be used to check conformance of traces from implementations against their specifications and flag violations. More recently, online model-based testing techniques have been developed to further strengthen conformance checking of implementations. However, a feature to generate a mock implementation from a ComMA specification is currently missing. The work in TechFlex 2025 addresses this gap.

3.3 Model-based Mocking with ComMA

This section is structured as follows: First, Section 3.3.1 provides a brief introduction to the syntax of the ComMA interface and component modeling languages. Then, Section 3.3.2 informally discusses the semantics of ComMA interfaces and highlights why trace semantics of ComMA components cannot be reused in an operational context. Section 3.3.3 presents a proposal to operationalize the semantics of component models. The intuition of such a semantics is informally presented using examples. Lastly, Section 3.3.4 discusses how a



⁵ [Eclipse CommaSuite](#)

component model is translated into executable C++ code, preserving the proposed semantics.

3.3.1 Introduction to ComMA specifications

Interfaces and Components are the two main concepts of in a ComMA specification.

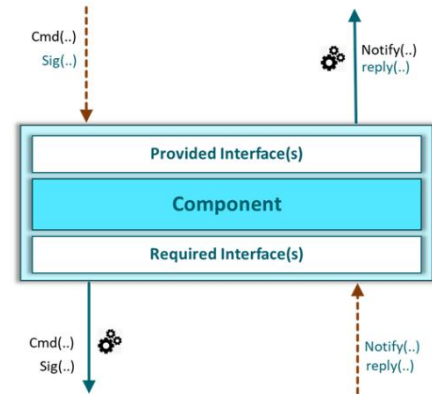
A ComMA interface model specifies all permissible sequences of interactions (i.e., assumed inputs and expected outputs) from the point of view of a service provider (server) towards its environment (clients).

Such a model consists of five main languages

1. *Types*: defines a set of data types built of simple types (int, real, bool, string, etc.), enumerations, vectors, records and maps
2. *Signature*: defines inputs (commands/signals), outputs (replies/notifications) and their associated types of data arguments. A command-reply models a synchronous call from a client to a server (client is blocked until reply from server), a signal models asynchronous innovation from client to server, while a notification models asynchronous invocation from server to client,
3. *Interface*: defines legal sequences of invocations between a client and server in terms of a protocol state machine. Such a state machine consists of a set of states (with an explicit initial) and transitions of two kinds, namely triggered and non-triggered. Triggered transitions model a sequence starting with either a command or signal (input), followed by a sequence of notifications or reply (outputs). Non-triggered transitions model a sequence of notifications or reply (outputs). To keep specifications compact, data variables may be declared, initialized and manipulated (updated) as part of a transition body using a pre-defined expressions language. Furthermore, transitions may define guards which are predicates over data variables and determine their enabling, i.e., whether a transition may be executed or not,
4. *Input parameter*: defines sets of data values corresponding parameters of commands/signals (provided interfaces) and notifications/replies (required interfaces) in each state they occur in the protocol state machine. Input parameters essentially define data values that are provided by a client in an operational context. These parameters are necessary to enable stand-alone simulation using only the server side state machine without requiring the modeler to explicitly specify the client side state machine. The assumption is that a user of a ComMA interface specification mirrors the state machine of the server.
5. *Constraints*: Next to a protocol state machine that defines permissible sequences of input and output interactions, timing and data requirements can be specified using the timing constraints and data constraints languages, which are part of the interface language but do not have an explicit relation to the transitions of a protocol state machine, rather to events listed in its signature. Timing constraints are of four types and specify admissible time intervals between events defined in the signature: (i) Interval rules constrain the allowed time interval for observing an event if a given preceding event was observed. (ii) conditional interval specifies that if two events are observed without observing the first event in between then the interval between them lies within a certain interval. (iii) periodic rules specify the time interval between recurring events, possibly with a jitter value. (iv) conditioned absence specifies that if a certain event is observed then another event must be absent during a given interval after the observation. The four types of rules can be combined in groups that form a scenario. Since we do not utilize

the data constraints language in the context of this report, we will omit its introduction.

A component rarely has a single interface. Typically a component cooperates with other components of the system to achieve an overall objective. Such cooperation by a component is achieved by offering services via its provided interfaces, and to fulfil those obligations may need to consume services of other components via its required interfaces. Clearly, relations need to be specified between events of provided and required interfaces. To model relations between interface events, *functional constraints* over protocol state machines of its interfaces can be specified in a *component* language. When no functional constraints are specified then it simply means that all possible interleavings of the interface state machines are allowed without any restrictions. Similar to an interface state machine, each functional constraint defines a state machine consisting of a set of states and transitions as edges between them, distinguished as triggered and non-triggered transitions. However the role of inputs and outputs have a slightly different meaning. Input triggers of a component correspond commands/signals of provided interfaces and notifications/replies of required interfaces, while outputs of a component are notifications/replies of provided interfaces and commands/signals of required interfaces.



3.3.2 Semantics and Challenges

The many ComMA languages define the syntax of interface and component specifications. The (execution) behavior or dynamics induced by such specifications is called its semantics. To prevent multiple or unintended interpretations of the same specification, as well as enable engineering automation (e.g., code generation, model-based testing, run-time monitoring, etc.), a formal semantics is essential. One way to give such semantics is by directly mapping each syntactic construct to a elements of a mathematical formalism such as Petri nets, automata or process algebra. Another way to give semantics is by defining the set of acceptable execution traces of an implementation of such a specification. The latter is a trace semantics which enables applications like run-time monitoring, while former is an operational semantics and enables a large variety of applications such as simulation, model checking, code synthesis (our focus), model-based testing, including run-time monitoring. The trace semantics of interface and component languages have been defined [51] and serve as basis for run-time monitoring.

The operational semantics of a ComMA interface specification is given by a straightforward mapping to colored Petri nets [52]. The kind of Petri nets defined by such a mapping belong to the well-studied classes [53] of state machines and open petri nets [54] with many interesting properties (for e.g., weak termination) that enable a variety of analysis [55] based on their structure. The mapping from ComMA syntax to colored Petri nets is implemented using the open-source SNAKES library [56] and serves as basis for enabling many applications:

- unambiguous documentation and interpretation of interface specifications
- simulation of interface specifications
- model quality checks to detect and notify modelers about potential design-errors, for e.g., races (leading to deadlocks), livelocks and confusions

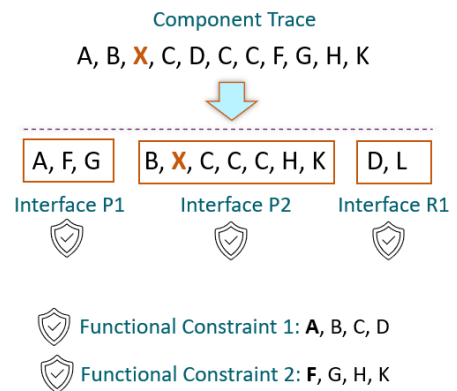
- automate test creation and execution by means of online model-based testing

The operational semantics of component models has not been defined. A direct reuse of the prescribed trace semantics [51] for purposes of generating executable artefacts such as mocks leads to many ambiguities as we will discuss in the next sub-section.

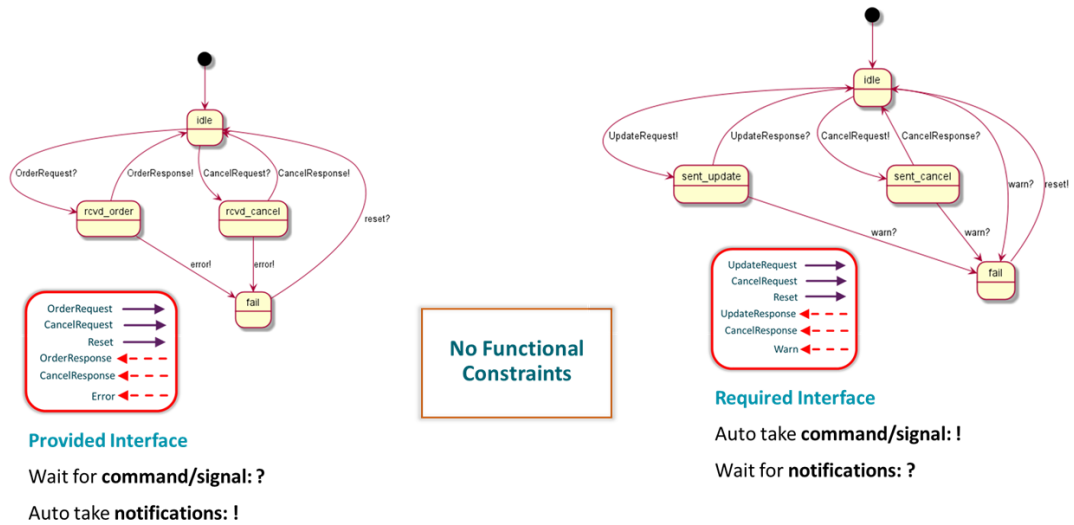
3.3.3 Informal Semantics of Components

The trace semantics of ComMA components defines a set of conforming traces without actually prescribing how to generate them. The trace set is very permissive, i.e., allows for significant non-determinism in the interleavings of interface and component state machines. Directly encoding such semantics as a set of synchronized state machines defining the behavior of generated mock implementations may lead to several ambiguities including deadlocks and termination problems, i.e., inability of component state machines to return back to initial state. The latter leads to situations where even though inputs from the environment allowed by the interface state machine, the component state machine is not able to accept them as they have not yet returned back to the initial state. To understand these issues better, consider the example shown in the Figure below.

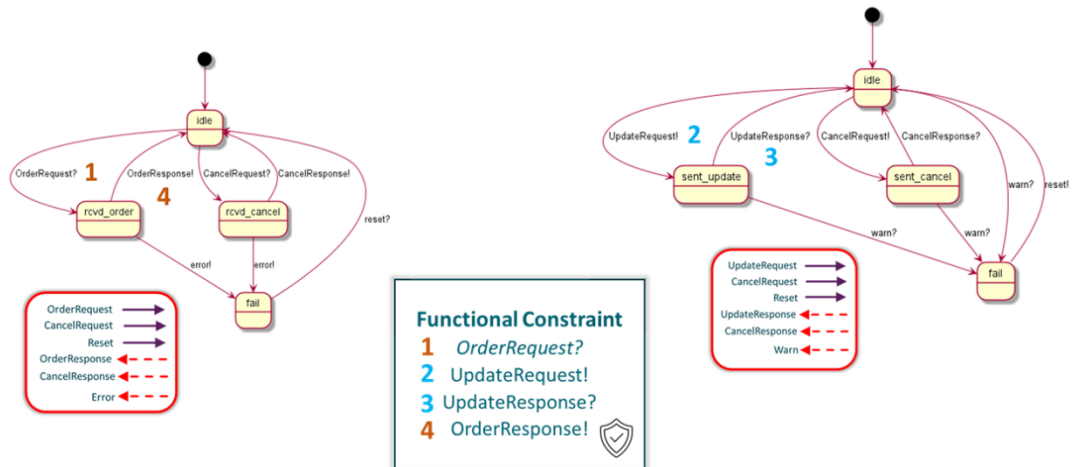
Here $A, B, C, D, F, G, H, K, L, X$ are the names of events of a component occurring in signatures of provided interfaces $P1$ (with events A, F, G), $P2$ (with events B, X, C, H, K), and required interface $R1$ (with events D, L). Furthermore, the ComMA interface specification of $P1$ allows only the trace AFG , $P2$ allows only the trace $BXCCCHK$ and $R1$ allows only the trace DL . Now suppose two functional constraints are defined over the signatures of $P1$, $P2$ and $R1$. The *functional constraint 1* allows the trace $ABCD$, which means that whenever an A is observed on $P1$, then events BCD must follow with no A, B, C, D in between. This means that events other than A, B, C, D may occur in between. This permissiveness leads to ambiguities in the operational semantics as we will see shortly. *Functional constraint 2* allows only the trace $FGHK$. Now suppose we are given a trace $ABXCDCCFGHK$ of a component implementing interfaces $P1$, $P2$ and $R1$. To check if this trace conforms to the ComMA specification described earlier, first a projection of this trace is taken over events of each interface (i.e., filter the trace by removing all events that do not belong to that interface) and checked that it is accepted, which turns out to be the case. Next for each functional constraint, a projection of the component trace is taken over the events used by it and checked if it is accepted. For both functional constraints it is easy to check that this is indeed the case.



Observe that after *functional constraint 1* becomes active (upon observing A), event X is allowed to occur at any moment since it does not belong to the set of used events: B, C, D . This means that interface events may interleave while a functional constraint has become active as long as they are accepted by the state machines of the interface and functional constraints. When we try to operationalize these semantics, we run into ambiguities as we will show in the next example. Consider the two interface state machines shown in the figure below, corresponding a provided (left) and required (right) interface. When no functional constraints are defined the behavior can be any interleaving of traces of the two state machines, i.e., the cross-product of the two state machines.



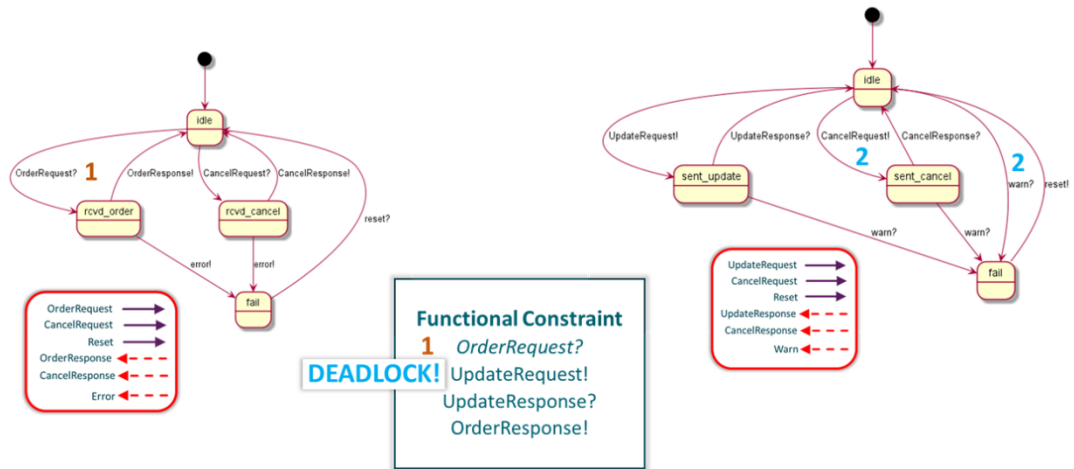
Now suppose a functional constraint is defined over the two interfaces. Then starting from the initial state of all state machines, one possible execution leads to a successful run.



However, according to trace semantics, an interface may decide to execute an event that is not part of the functional constraint after it has become active. Suppose after *OrderRequest* was observed on provided interface, the required interface executed *CancelRequest*, leading to state *sent_cancel*. In this state, the functional constraint *deadlocks* as it is no longer possible to execute *UpdateRequest* on the required interface as specified, until either the sequence with events *CancelResponse* or *Warn*, *Reset* is executed by the required interface. But executing these sequences is ambiguous, when there are other functional constraints using these events.

For instance, suppose there is another functional constraint *F* that states when a *CancelRequest* is received on provided interface then *CancelRequest* must be executed on the required interface, followed by waiting for a *CancelResponse* on the required interface, and only then a *CancelResponse* on the provided interface.

Going back to our case of the deadlocked functional constraint, executing *CancelRequest* on the required interface, after receiving *UpdateRequest* violates the functional intent of *F*, which was to allow *CancelRequest* on required only if a *CancelRequest* was observed on provided.



To prevent ambiguities arising from interleaving events of more than one functional constraint requires a reduction in the allowed non-determinism in semantics of functional constraints in conjunction with what is allowed to be executed on interface state machines. As a solution, the following restrictions on interpretation of ComMA component specifications are proposed

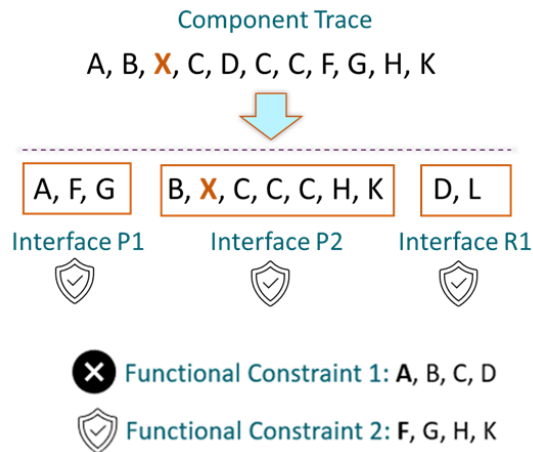
Syntax

- **R1.** Only the initial state of a functional constraint may contain triggered transitions.
- **R2.** An initial state may not contain any non-triggered transitions.
- **R3.** The state machine of a functional constraint is strongly connected.
- **R4.** In addition to local variables of functional constraints, global variables may be declared and shared between functional constraints.

Semantics

- **R5.** A functional constraint is enabled if it is in its initial state, receives a trigger event specified by one of its triggered transitions, and satisfies the guard predicate.
- **R6.** When an enabled functional constraint executes its trigger event in synchronization with the corresponding interface state machine, it becomes active.
- **R7.** An active functional constraint executes an allowed sequence of outgoing events synchronously with their corresponding interface state machines, until the function constraint state machine returns back to its initial state. The transition back to its initial state, additionally resets its local variables and only then the function constraint becomes deactivated.
- **R8.** When a functional constraint is active, no interface state machines may execute a transition on their own. This restriction means that functional constraints, once triggered, *run-to-completion*.
- **R9.** When no functional constraints are active, interface state machines may execute their transitions in the following manner: provided interfaces may execute allowed non-triggered transitions, while required interfaces may execute allowed triggered transitions.
- **R10.** At most one functional constraint may be active at any given moment.

As a consequence of the above restrictions, *functional constraint 1* from our earlier example is violated and no longer allowed under the new restrictions.

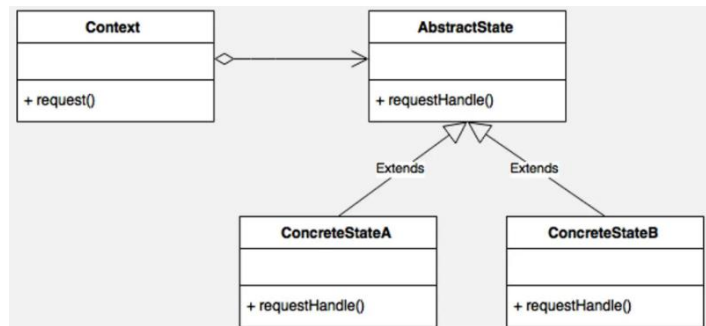


A formalization of the above operational semantics of ComMA component language is one of the main goals of TechFlex 2026.

3.3.4 Designing a Generator of C++ Component Mocks

Our goal to generate a mock implementation from a given ComMA specification, requires us to encode the semantics as synchronous interface and component state machines in a programming language of choice. We choose the C++ programming language as it is a de-facto standard for software implementations in the domain of cyber-physical systems.

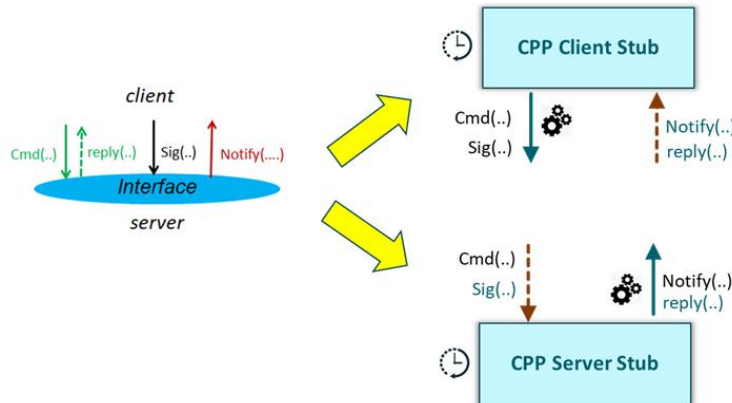
Realizing state machines in programming languages is a well-known concept. As early as the 1990s, design patterns were proposed [57] as good practices to tackle recurring implementation problems encountered in practice using object oriented principles. Among them the *state machine pattern* was proposed. The pattern exploits the notion of an abstract state class containing an execution handler function. Each concrete instance of this class implements the handler function, each defining specific actions to be performed in that state. The destruction and creation of concrete state objects correspond to transition between states of a state machine. Such an implementation pattern avoids complex conditional logic by encapsulating behavior within separate state classes instead of conditional logic such as if-else/switch cases. Each state independently handles its own transitions and actions, reducing code duplication and branching complexity, resulting in modular code that is easier to extend and maintain as new states are added.



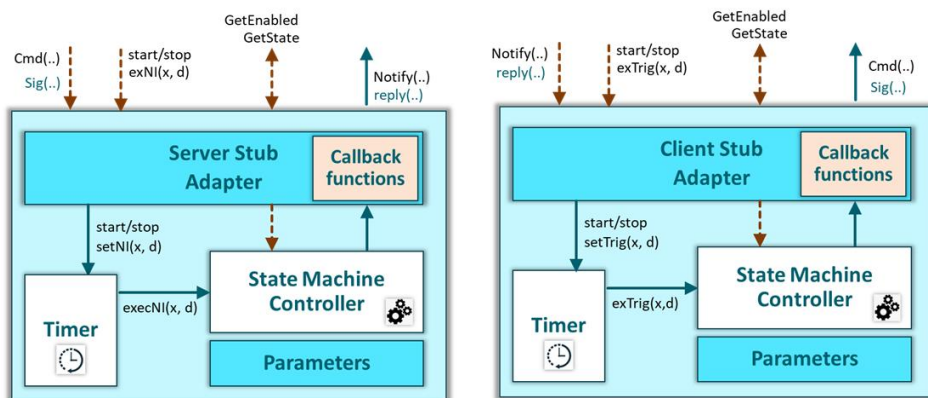
Realizing state machines in programming languages is a well-known concept. As early as the 1990s, design patterns were proposed [57] as good practices to tackle recurring implementation problems encountered in practice using object oriented principles. Among them the *state machine pattern* was proposed. The pattern exploits the notion of an abstract state class containing an execution handler function. Each concrete instance of this class implements the handler function, each defining specific actions to be performed in that state. The destruction and creation of concrete state objects correspond to transition between states of a state machine. Such an implementation pattern avoids complex conditional logic by encapsulating behavior within separate state classes instead of conditional logic such as if-else/switch cases. Each state independently handles its own transitions and actions, reducing code duplication and branching complexity, resulting in modular code that is easier to extend and maintain as new states are added.

To synthesize mock implementations in C++ from ComMA specifications, we need to encode the semantics discussed in the previous section. To do so, we first introduce the notion of an server and client interface stubs, corresponding provided and required interfaces, respectively. Both these stubs are realized using the state machine pattern with key differences to satisfy requirement *R9*. From the current state, a server interface stub may autonomously execute a possible notification/reply (to command) but requires an external trigger to be received to execute a command or signal, if they are allowed in that state. If not,

then an error is thrown and the implementation exits. In a client interface stub, the roles are reversed.

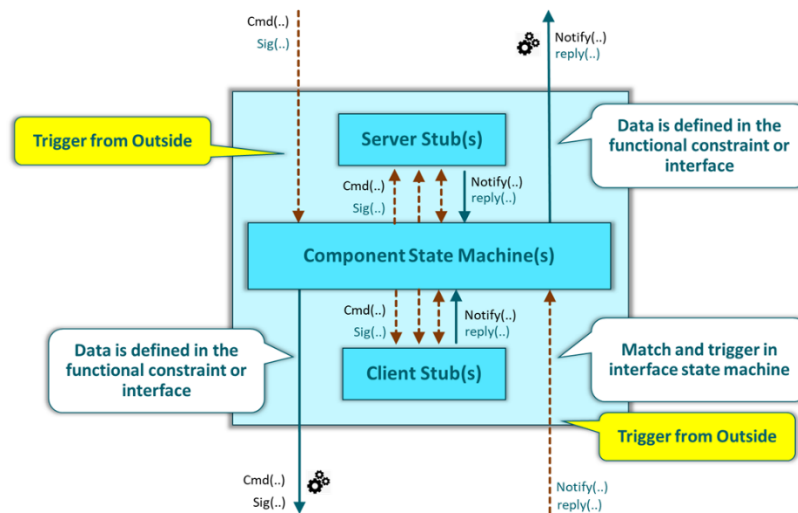


A detailed implementation view of server and client interface stubs are shown below. Observe that to handle external triggers, the server interface stub accepts *Cmd(...)* and *Sig(...)*, while the client interface stub accepts *Notify(...)* and *reply(...)*. To synchronize execution of events of a functional constraint with an interface state machine, it is necessary to check the current state of an interface state machine and whether an event is possible in that state. The method *GetEnabled* of server/client interface stubs returns the list of enabled transitions. Furthermore, the method *GetState* returns the name of the current state of the interface state machine, which is needed to check enabling of a guard of a functional constraint which may require an interface to be in a certain state. The two methods, provide an implementation of the functional constraint state machine controller with the means to realize *R5* and *R7*.

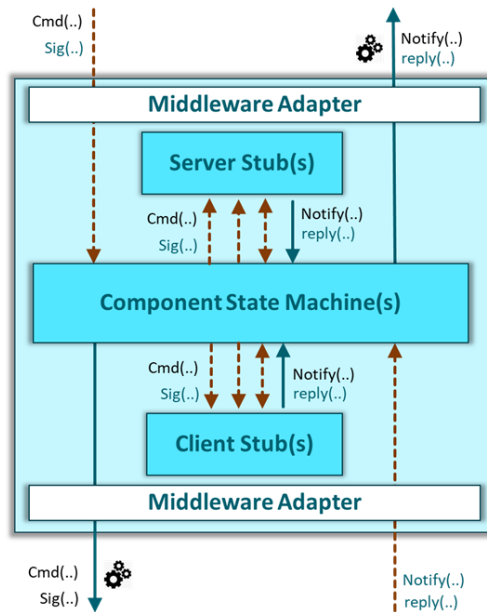


The timer component of interface stubs, enables the autonomous behavior as prescribed by *R9*, i.e., whenever the timer goes off (pre-defined intervals), each stub type is triggered to execute an event (at random) that is allowed to be taken in that state, on their own initiative. The methods *start* and *stop* control the enabling and disabling of the timer which allows the implementation of a functional constraint state machine to prevent interface state machines from executing events when a functional constraint is active (*R8* and *R9*). When a functional constraint has become active, the execution of a sequence of outgoing events synchronously with their corresponding interface state machines (*R7*) is made possible by the methods *execNI(x,d)* and *execTrig(x,d)*, where *x* is the name of the event and *d* is a set of data parameters of *x*.

An illustration of an implementation of a component state machine (corresponding to a set of functional constraints) orchestrating a set of server and client interface stubs is shown below. Incoming trigger events to a component are handled by the component state machine. If a functional constraint is defined then the guards are evaluated to determine if it becomes active. When a functional constraint is active then it immediately synchronizes the execution of the incoming trigger event and the successive events (run-to-completion) defined by the functional constraint with the corresponding interface state machine (stub). If no functional constraint is defined for the trigger then it is simply forwarded to the interface state machine (stub). If an interface state machine (stub) cannot make a synchronized move with a functional constraint state machine because that event is not possible in the current state then the application crashes with an exception stating that an illegal event was received. Note that incoming triggers carry relevant data arguments which are inputs, whereas outgoing events from a component contain data that are computed by the expressions defined in a functional constraint or interface state machine specification.

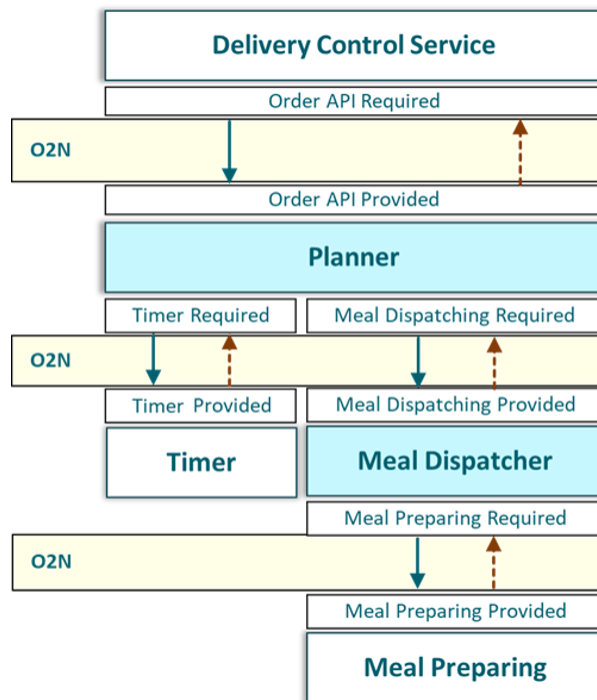


An additional requirement for the generation of C++ component mocks was to also generate glue code to integrate with specific middleware technologies. For this an adapter layer is also generated to avoid manual intervention by a developer.



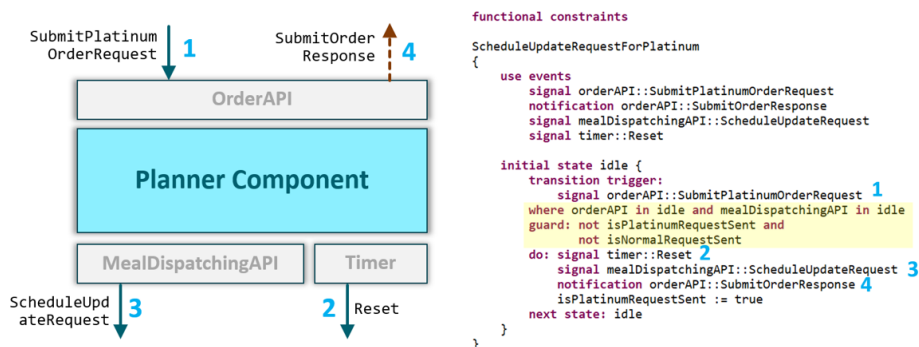
3.4 Application to the Meal Delivery System

The use case we used to show the feasibility of synthesizing mock components from ComMA specifications is a meal delivery system, as sketched below. There are five main components: delivery control service, planner, timer, meal dispatcher and meal preparing. The components colored in blue represent the ones that have at least one functional constraint. The ones without any color do not specify any functional constraint, i.e., they rely on the behavior specified by their interface state machines. The specific middleware layer chosen to support communication between components for the demonstrator is a proprietary solution called O2N.



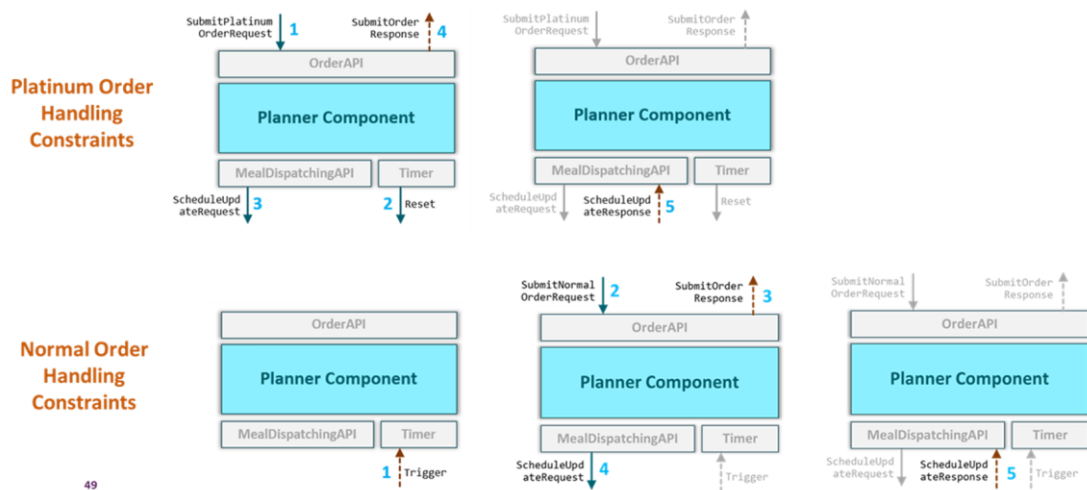
Planner Component: The planner component is responsible for handling normal and platinum order requests generated by the delivery service component. Depending on the type of request, a sequence of outputs consisting of signals to the meal dispatcher and notifications to the delivery control service are generated by the planner component. The timer component issues periodic triggers to the planner component to process any outstanding normal order requests.

The functional constraint modeling the behavior of a platinum order request is shown below. It specifies a state machine with a single state called *idle* and a transition that returns back to *idle* (see *next state*). The transition is triggered by the receipt of a *SubmitPlatinumOrderRequest* on the *orderAPI* interface if the following *conditions* hold: *OrderAPI* and *MealDispatchingAPI* are in their *idle* states and the variables *isPlatinumRequestSent* and *isNormalRequestSent* evaluates to *false* (highlighted in yellow). If the conditions are satisfied then the following events are executed in a sequence (without interleaving with any other events): first a *reset* on the *timer* interface, then a *ScheduleUpdateRequest* on *MealDispatchingAPI* and *SubmitOrderResponse* on *OrderAPI*, and finally assigning the variable *isPlatinumRequestSent* to *true*.



To conform to run-to-completion semantics which require that a functional constraint may contain at most one input trigger, the handling of the platinum order request is split over two functional constraints synchronizing over a shared variable *isPlatinumRequestSent*. The second functional constraint handles the trigger *ScheduleUpdateResponse* generated by the meal dispatcher component with a guard on variable *isPlatinumRequestSent*.

Note that to model normal order request, three functional constraints were needed. The first one (starting from left in the figure) models the receipt of a trigger from the timer. The second functional constraint is triggered by the arrival of a *ScheduleNormalOrderRequest*, and as a consequence generates *ScheduleOrderResponse* towards the *delivery control service* component and *ScheduleUpdateRequest* towards the *meal dispatcher* component. The third functional constraint models the receipt of *ScheduleUpdateResponse* generated by the meal dispatcher component, similar to the functional constraint modeling platinum order request.



Two examples of timing constraints of the Planner component is shown below.

```
group PlatinumOrderRequestResponse
  signal orderAPI::SubmitPlatinumOrderRequest
  -[500.0 ms .. ]-> signal mealDispatchingAPI::ScheduleUpdateRequest
  -[ .. 0.01 ms]-> notification orderAPI::SubmitOrderResponse
end group

NormalOrderRequestResponse
  signal orderAPI::SubmitNormalOrderRequest
  -[ .. 20.0 ms]-> notification orderAPI::SubmitOrderResponse
```

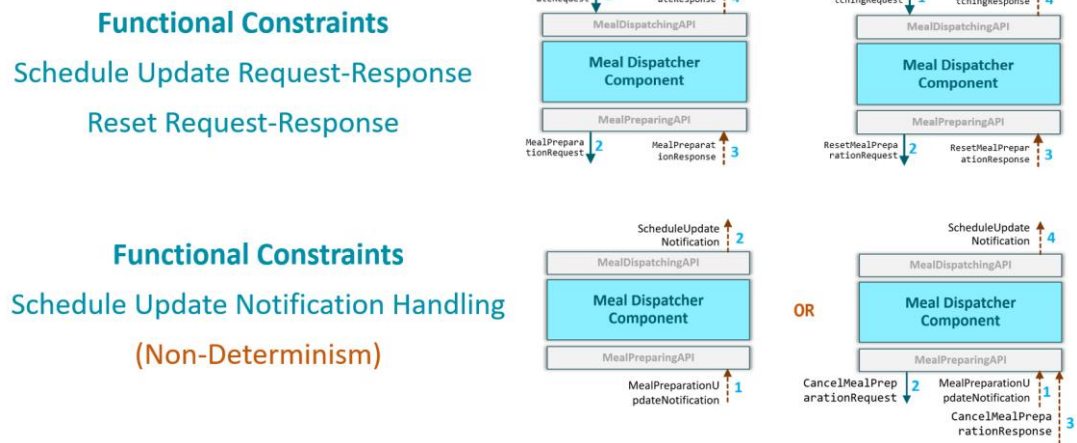
The first one states that after a *SubmitPlatinumOrderRequest* is received, at least 500 ms must have passed before a *ScheduleUpdateRequest* is send to the meal dispatcher component, and at most 0.01 ms may have passed before generating a *SubmitOrderResponse* notification.

The second one states that after a *SubmitNormalOrderRequest* is received, at most 20 ms may pass before a *SubmitOrderResponse* is generated.

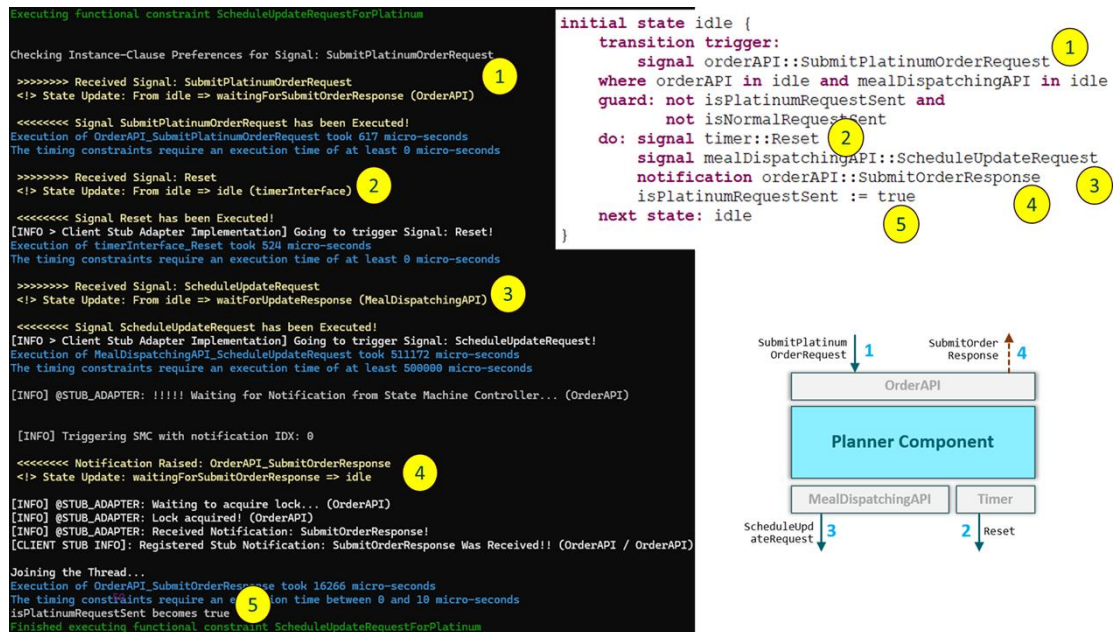
Meal Dispatching Component

In a similar manner to the Planner component, four functional constraints of the meal dispatcher component are defined. An illustration of the specified behavior is shown in the figure below. An interesting case shown in this example is non-determinism between functional constraints. In response to the trigger *MealPreparationUpdateNotification*

generated by the meal preparing component, two function constraints are defined with no specific guards. As a result they may both be enabled and then the choice of their execution is non-deterministic. This allows a modeler to under-specify behavior enabling to define possible and allowed behaviors to occur at random.



Results. In TechFlex 2025, we developed proof-of-concept code generators to synthesize C++ mocks over O2N middleware from ComMA component models. The feasibility of this approach was demonstrated using the example of the Meal Deliver System to stakeholders at Thales. A snapshot from the demonstrator depicting console output from running the planner component is shown below. The console output corresponds to the run-to-completion semantics expressed in functional constraint, i.e., when a *SubmitPlatinumOrderRequest* comes in as a trigger (highlighted as 1), it is first checked to evaluate if the guard and where clauses are satisfied, and only then synchronizes with MealDispatchingAPI interface to execute the trigger. Now the functional constraint has become active, so it runs to completion by executing the events of the do section: *reset* (highlighted as 2), *ScheduleUpdateRequest* (highlighted as 3) and *SubmitOrderResponse* (highlighted as 4), followed by setting variable *isPlatinumRequestSent* to *true*. Note that the timing durations between these events conform to the timing constraints defined earlier.



The focus of TechFlex 2025 has been to cover control-flow aspects of the ComMA language. As a next step in TechFlex 2026, we intend to extend support for data arguments of events and timing constraints. Furthermore, the work on defining generators for C++ mocks raised many interesting questions and provided practical insights into corner cases which have helped to define informally the operational semantics of the ComMA component language. The formalization of these semantics will be addressed in TechFlex 2026 to enable model checking capabilities that ensure specifications are free of modeling and design errors.

4 Conclusions

This chapter concludes the report by first answering the research question in Section 4.1, followed by a discussion about future work in Section 0.

4.1 Answers to the Research Questions

RQ1) [Automatic synthesis of product variants based on functionality and constraints]

- To what extent is it possible to synthesize from a product (platform) variation tree, software configurations that satisfy both functionality requirements, constraints, and performance requirements, while optimizing other system qualities such as cost?

Chapter 2 shows that it is indeed feasible to automatically synthesize software configurations for new product variants that satisfy the functional and non-functional requirements. The key enabler is a model-based system synthesis approach in which reusable components are described in terms of the services they provide and require, their resource needs (CPU, memory), and properties, such as temporal behavior or the quality of the provided output.

Given a high-level specification of the required system services and properties, these component descriptions form the basis for expressing system synthesis as an optimization problem. The synthesis tool searches for valid combinations of components and hardware deployments that collectively provide all required functionality. Because many candidate configurations may exist, the synthesis is guided by an objective function (e.g., minimizing hardware cost or maximizing performance).

To address industrial needs, a new synthesis methodology is proposed, building on the existing ADMORPH approach [27]. In particular, this methodology is extended to make it applicable to complex systems that are often hierarchically decomposed into subsystems, possibly in several steps, and deployed on tens or hundreds of compute nodes organized in racks and connected using multiple networks. Evaluations on both synthetic and industrial cases demonstrate that this extended method can efficiently generate valid system configurations in seconds or minutes.

RQ2) [Component specification is required for early system performance analysis]

- What defines a (minimal) set of component behavior and interface specifications needed to generate an executable system model that can be used to validate system flows and their performance in an early phase without the need of a complete model of all components?

Chapter 3 demonstrates that early system-level validation and performance analysis can be enabled by generating executable mock components from high-level interface and component models. Using ComMA, *partial* aspects of a component behavior can be effectively specified from a black box perspective in terms of interface protocols and a set of functional constraints. Such specification provide a precise yet lightweight basis for mock generation without requiring a full specification covering all aspects of an implementation. To make these specifications executable, an operational semantics was defined on top of

existing trace semantics, restricting the degree of non-determinism (e.g., through run-to-completion semantics). These restriction were essential to eliminate ambiguities that could otherwise lead to deadlocks and other termination problems. Based on this foundation, the approach enables the automatic generation of behaviorally consistent mock implementations that can be deployed on target platforms. The demonstrator validates that this approach is feasible and will enable system designers and architects to detect timing issues and platform-induced effects earlier, even before the full software implementations are made available by development teams.

RQ3) [Incremental performance prediction for product families]

- To what extent is it possible to accurately predict performance of arbitrary product configurations based on incremental insights for individual components gained during development and deployment?

The answer to this question is based on the work in [11], which was conducted as part of this research. The key to predicting performance for arbitrary product configurations and deployments in a scalable way is to use a compositional performance analysis. This means that performance characteristics are collected for individual components and that there is a framework that allows reasoning about their performance when deployed on shared resources, e.g., a multi-core CPU with shared memory.

To accurately predict performance degradation caused by memory contention, [8] proposes a structured three-step methodology that extends the Bubble-Up approach [58]. In this method, each application is individually characterized in two different ways: 1) sensitivity to memory interference. This is achieved by deploying the application with a tunable synthetic Source of Interference (Sol) that progressively increases memory pressure. The resulting measurements form a sensitivity curve, capturing how the application's latency changes as contention intensifies. 2) The contentiousness of the application on the memory resource, how much pressure it puts on the resource itself, is determined using a "reporter", a representative proxy application with known sensitivity curve and high responsiveness to memory pressure.

Finally, performance degradation for a pair of co-running applications is predicted by combining the target application's sensitivity curve and the competitor's contentiousness score. This yields a direct estimate of the expected slowdown without requiring actual co-deployment, making it a compositional approach. The compositional nature of the approach makes it suitable for environments where insights about components and their performance are achieved incrementally as they are developed and evolve. Before a component has been fully implemented, sensitivity curves and contentiousness scores have to be estimated based on experience from previous components. However, as soon as a component has been implemented—and every time it evolves—the component can be quickly characterized, and its performance can be estimated for arbitrary deployments.

Overall, the proposed method shows strong predictive accuracy when evaluated with (median absolute error $\approx 1.4\%$, 90th percentile absolute error $\approx 7.4\%$), demonstrating its efficacy for performance prediction of microservice deployments on multi-core platforms.

Together, the results of the project and answers to the three research questions show that the proposed approach addresses the three challenges identified in the introduction. They reduce the dependence on scarce expert knowledge by making component selection and its impact on system qualities more explicit, support higher engineering productivity by automating parts of the integration process, and lower the effort of evolving product variants by using technology-agnostic abstractions as models to capture and synthesize expected system behavior and support validation against updated or changed requirements.

4.2 Future Work

Future work in the area of synthesis of component-based systems (RQ1) involves maturing the generalized synthesis process and associated tool to make it easy to use and to visualize synthesis results for complex systems in a scalable way. Once mature, it is important to evaluate the quality of synthesized solutions, defined by the objective function, to baseline solutions that are manually created by engineers to determine whether optimized solutions are actually better than the base line. The approach also needs to be evaluated in terms of its effect on engineering productivity to clearly determine the value of the approach in practice.

There are many possible avenues for future research related to synthesis of component-based systems. Ideas include investigating whether general design rules can be extracted from the many visited design points, to go beyond presenting an optimized solution to presenting insights about what implementations of subsystems or what deployments that generally seem to work well and which ones do not. This would allow engineers to learn about the systems they build, as well as the building blocks they use to construct them, and increase the trust in synthesized solutions.

For the synthesis of mock components (RQ2) to enable early system performance analysis in practice, future work will focus on extending the proof-of-concept generator to support data and timing aspects. Next to this the focus will also be on formalizing the semantics of the ComMA component language. This will enable the development of features to improve engineering efficiency and effectiveness such as model checking to detect early any design errors, model-based testing and run-time monitoring to automate conformance testing of components against their specifications.

For performance prediction (RQ3), future work focuses on extending the prediction methodology beyond pairwise co-locations, enabling performance estimation for larger groups of concurrently running applications. This could be done e.g. by using the scalability approach proposed in [59]. Additionally, the prediction approach could be broadened to model contention in other shared resources, e.g. CPU, network, and disk, e.g. requiring new resource-specific sources of interference and reporters. Another direction is adapting the method to heterogeneous hardware platforms, where differences in memory hierarchies and execution behavior may affect prediction accuracy.

References

- [1] T. Hendriks and S. Acur, "Vision and Outlook for Systems Architecting and Systems Engineering in the High-Tech Equipment Industry," TNO 2024 R10542, 25 2024. Accessed: Oct. 21, 2024. [Online]. Available: <https://repository.tno.nl/SingleDoc?find=UID%2091ca077f-403c-4589-9d04-da0695f463e6>
- [2] A. van der Werf, G. Upis, I. Oehlmann, J. W. Velthuisen, J. Castillo, and R. Ribas Santolim, "Made in NL: The value of the Dutch high-tech manufacturing industry," PricewaterhouseCoopers, Jan. 2024. Accessed: Mar. 21, 2025. [Online]. Available: <https://www.pwc.nl/en/insights-and-publications/services-and-industries/industrial-manufacturing/hightech-industry-of-great-value-to-the-dutch-economy.html>
- [3] B. Akesson, B. Pronk, S. Weiss, and D. Worrel, "Software Evolution of Product Families in High-tech Equipment," TNO, TNO 2025 R10901, 2025.
- [4] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Sci. Comput. Program.*, vol. 89, pp. 144–161, Sep. 2014, doi: 10.1016/j.scico.2013.03.017.
- [5] J. Aranda, D. Damian, and A. Borici, "Transition to Model-Driven Engineering," in *Model Driven Engineering Languages and Systems*, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., Berlin, Heidelberg: Springer, 2012, pp. 692–708. doi: 10.1007/978-3-642-33666-9_44.
- [6] H. Burden, R. Heldal, and J. Whittle, "Comparing and contrasting model-driven engineering at three large companies," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, in ESEM '14. New York, NY, USA: Association for Computing Machinery, Sep. 2014, pp. 1–10. doi: 10.1145/2652524.2652527.
- [7] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Softw. Syst. Model.*, vol. 17, no. 1, pp. 91–113, Feb. 2018, doi: 10.1007/s10270-016-0523-3.
- [8] B. Dzikowski, "Practical Recommendations for Accurately Predicting Performance Degradation Caused by Memory Contention," University of Amsterdam, 2025.
- [9] V. Giedrimas, S. Omanovic, and P. Grigorenko, "The evolution of automated component-based software development tools: From structural synthesis of programs to behavioral types," in *2017 International Conference on Information Science and Communications Technologies (ICISCT)*, Nov. 2017, pp. 1–6. doi: 10.1109/ICISCT.2017.8188570.
- [10] P. Bartalos and M. Bieliková, "Effective QoS Aware Web Service Composition in Dynamic Environment," in *Information Systems Development*, J. Pokorny, V. Repa, K. Richta, W. Wojtkowski, H. Linger, C. Barry, and M. Lang, Eds., New York, NY: Springer, 2011, pp. 101–113. doi: 10.1007/978-1-4419-9790-6_8.
- [11] "ADMORPH Project Deliverable D5.3 – Report on use cases and full prototype demonstrators," Confidential, 31 2023.
- [12] F. van den Berg *et al.*, "QRML: A Component Language and Toolset for Quality and Resource Management," in *2020 Forum for Specification and Design Languages (FDL)*, Sep. 2020, pp. 1–8. doi: 10.1109/FDL50818.2020.9232936.
- [13] D. Schuller, A. Polyvyanyy, L. García-Bañuelos, and S. Schulte, "Optimization of Complex QoS-Aware Service Compositions," in *Service-Oriented Computing*, G. Kappel, Z. Maamar, and H. R. Motahari-Nezhad, Eds., Berlin, Heidelberg: Springer, 2011, pp. 452–466. doi: 10.1007/978-3-642-25535-9_30.

- [14] G. Mints and E. Tyugu, "Justification of the structural synthesis of programs," *Sci. Comput. Program.*, vol. 2, no. 3, pp. 215–240, Dec. 1982, doi: 10.1016/0167-6423(83)90016-3.
- [15] P. Grigorenko, A. Saabas, and E. Tyugu, "COCOViLA – Compiler-Compiler for Visual Languages," *Electron. Notes Theor. Comput. Sci.*, vol. 141, no. 4, pp. 137–142, Dec. 2005, doi: 10.1016/j.entcs.2005.05.009.
- [16] V. Giedrimas, "Towards the Curry-Howard protocol implementation for the component-based paradigm," in *2014 IEEE 2nd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, Nov. 2014, pp. 1–5. doi: 10.1109/AIEEE.2014.7020320.
- [17] "CoCoViLa Tool." Accessed: Nov. 20, 2025. [Online]. Available: <https://cocovila.github.io/>
- [18] M. Hendriks, M. Geilen, K. Goossens, R. de Jong, and T. Basten, "Interface Modeling for Quality and Resource Management," *Log. Methods Comput. Sci.*, vol. Volume 17, Issue 2, May 2021, doi: 10.23638/LMCS-17(2:19)2021.
- [19] "Xtext - Language Engineering Made Easy!" Accessed: Nov. 20, 2025. [Online]. Available: <https://eclipse.dev/Xtext/>
- [20] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman, "Integrating Interface Modeling and Analysis in an Industrial Setting:," in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2017, pp. 345–352. doi: 10.5220/0006133103450352.
- [21] I. Kurtev, J. Hooman, M. Schuts, and D. van der Munnik, "Model based component development and analysis with ComMA," *Sci. Comput. Program.*, vol. 233, p. 103067, Mar. 2024, doi: 10.1016/j.scico.2023.103067.
- [22] "Z3," Microsoft Research. Accessed: Nov. 20, 2025. [Online]. Available: <https://www.microsoft.com/en-us/research/project/z3-3/>
- [23] M. Geilen, T. Basten, B. Theelen, and R. Otten, "An Algebra of Pareto Points," *Fundam. Informaticae*, vol. 78, no. 1, pp. 35–74, Jun. 2007, doi: 10.3233/FUN-2007-78103.
- [24] "Quality and Resource Modeling Language – A site about QRML, the Quality and Resource Modeling Language." Accessed: Nov. 20, 2025. [Online]. Available: <https://qrml.org/static/>
- [25] S. T. Nikkhah, M. Geilen, D. Goswami, M. Koedam, A. Nelson, and K. Goossens, "A Deployment Framework for Quality-Sensitive Applications in Resource-Constrained Dynamic Environments," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, Sep. 2021, pp. 212–220. doi: 10.1109/DSD53832.2021.00042.
- [26] "IBM ILOG CPLEX Optimization Studio." Accessed: Nov. 07, 2024. [Online]. Available: <https://www.ibm.com/products/ilog-cplex-optimization-studio>
- [27] "ADMORPH Project," ADMORPH. Accessed: Dec. 02, 2025. [Online]. Available: <http://admorph.eu/>
- [28] R. Auge, "Using Requirements and Capabilities." Accessed: Dec. 02, 2025. [Online]. Available: <https://blog.osgi.org/2015/12/using-requirements-and-capabilities.html>
- [29] D. Scheurink, "Constraint based resolving for dynamic systems," NHL Stenden University of Applied Sciences, 2019.
- [30] *DEAP*. (Dec. 02, 2025). Python. Distributed Evolutionary Algorithms in Python. Accessed: Dec. 02, 2025. [Online]. Available: <https://github.com/DEAP/deap>
- [31] M. Vollaard, "Hardware Dimensioning for Microservice-based Cyber-Physical Systems: A Profiling and Performance Prediction Method," Vrije Universiteit Amsterdam, 2024.
- [32] M. Alharbi and M. Alshayeb, "Automatic Code Generation Techniques: A Systematic Literature Review," *Autom. Softw. Eng.*, vol. 33, no. 1, p. 4, Sep. 2025, doi: 10.1007/s10515-025-00551-3.
- [33] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*:

- Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds., Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. doi: 10.18653/v1/N19-1423.
- [34] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, “The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming,” in *Proceedings of the 24th Australasian Computing Education Conference*, in ACE ’22. New York, NY, USA: Association for Computing Machinery, Feb. 2022, pp. 10–19. doi: 10.1145/3511861.3511863.
- [35] J. Y. Khan and G. Uddin, “Combining Contexts from Multiple Sources for Documentation-Specific Code Example Generation,” in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2023, pp. 683–687. doi: 10.1109/SANER56733.2023.00071.
- [36] F. Pinto-Santos, Z. Alizadeh-Sani, D. Alonso-Moro, A. González-Briones, P. Chamoso, and J. M. Corchado, “A Template-Based Approach to Code Generation Within an Agent Paradigm,” in *Highlights in Practical Applications of Agents, Multi-Agent Systems, and Social Good. The PAAMS Collection*, F. De La Prieta, A. El Bolock, D. Durães, J. Carneiro, F. Lopes, and V. Julian, Eds., Cham: Springer International Publishing, 2021, pp. 296–307. doi: 10.1007/978-3-030-85710-3_25.
- [37] A. D. Durai, M. Ganesh, R. M. Mathew, and D. K. Anguraj, “A novel approach with an extensive case study and experiment for automatic code generation from the XML schema Of UML models,” *J. Supercomput.*, vol. 78, no. 6, pp. 7677–7699, Apr. 2022, doi: 10.1007/s11227-021-04164-x.
- [38] T. Kaluarachchi and M. Wickramasinghe, “WebDraw: A machine learning-driven tool for automatic website prototyping,” *Sci. Comput. Program.*, vol. 233, p. 103056, Mar. 2024, doi: 10.1016/j.scico.2023.103056.
- [39] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, “Template-based Neural Program Repair,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, May 2023, pp. 1456–1468. doi: 10.1109/ICSE48619.2023.00127.
- [40] Z. Yang *et al.*, “Improving domain-specific neural code generation with few-shot meta-learning,” *Inf. Softw. Technol.*, vol. 166, p. 107365, Feb. 2024, doi: 10.1016/j.infsof.2023.107365.
- [41] D. Apostol, P. Rusovan, and M. Marcu, “UML to code, and code to UML, a view inside implementation challenges and cost,” in *2022 26th International Conference on System Theory, Control and Computing (ICSTCC)*, Oct. 2022, pp. 140–145. doi: 10.1109/ICSTCC55426.2022.9931871.
- [42] Q. Long, Z. Liu, X. Li, and H. Jifeng, “Consistent code generation from UML models,” in *2005 Australian Software Engineering Conference*, Mar. 2005, pp. 23–30. doi: 10.1109/ASWEC.2005.17.
- [43] M. Hussein, R. Nouacer, A. Radermacher, A. Puccetti, C. Gaston, and N. Rapin, “An end-to-end framework for safe software development,” *Microprocess. Microsyst.*, vol. 62, pp. 41–49, Oct. 2018, doi: 10.1016/j.micpro.2018.07.004.
- [44] E. Seidewitz, “UML with meaning: executable modeling in foundational UML and the Alf action language,” in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, in HILT ’14. New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 61–68. doi: 10.1145/2663171.2663187.
- [45] F. Ciccozzi, “On the automated translational execution of the action language for foundational UML,” *Softw. Syst. Model.*, vol. 17, no. 4, pp. 1311–1337, Oct. 2018, doi: 10.1007/s10270-016-0556-7.
- [46] L. Ma, S. Gui, L. Luo, and L. Yin, “Research of Automatic Code Generating Technology Based on AADL,” in *2008 International Conference on Embedded Software and Systems Symposia*, Jul. 2008, pp. 136–141. doi: 10.1109/ICESS.Symposia.2008.49.

- [47] S. Gui, L. Ma, L. Luo, L. Yin, and Y. Li, "UCaG: An Automatic C Code Generator for AADL Based upon DeltaOS," in *2008 International Conference on Advanced Computer Theory and Engineering*, Dec. 2008, pp. 346–350. doi: 10.1109/ICACTE.2008.68.
- [48] D. Harel, "Statecharts: a visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987, doi: 10.1016/0167-6423(87)90035-9.
- [49] S. Van Mierlo and H. Vangheluwe, "Introduction to Statecharts Modeling, Simulation, Testing, and Deployment," in *2019 Winter Simulation Conference (WSC)*, Dec. 2019, pp. 1504–1518. doi: 10.1109/WSC40007.2019.9004771.
- [50] M. Chiari, M. Camilli, M. M. Bersani, R. van Beusekom, and D. A. Tamburri, "Reality Check on Formal Methods in Industry: A Study of Verum Dezyne," *J. Softw. Evol. Process*, vol. 37, no. 12, p. e70069, 2025, doi: 10.1002/smr.70069.
- [51] I. Kurtev, J. Hooman, and M. Schuts, "Runtime Monitoring Based on Interface Specifications," in *ModelEd, TestEd, TrustEd: Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, J.-P. Katoen, R. Langerak, and A. Rensink, Eds., Cham: Springer International Publishing, 2017, pp. 335–356. doi: 10.1007/978-3-319-68270-9_17.
- [52] K. Jensen, "Coloured Petri nets," in *Petri Nets: Central Models and Their Properties*, W. Brauer, W. Reisig, and G. Rozenberg, Eds., Berlin, Heidelberg: Springer, 1987, pp. 248–299. doi: 10.1007/BFb0046842.
- [53] K. van Hee, N. Sidorova, and M. Voorhoeve, "Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach," in *Applications and Theory of Petri Nets 2003*, W. M. P. van der Aalst and E. Best, Eds., Berlin, Heidelberg: Springer, 2003, pp. 337–356. doi: 10.1007/3-540-44919-1_22.
- [54] K. Wolf, "Does My Service Have Partners?," in *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems*, K. Jensen and W. M. P. van der Aalst, Eds., Berlin, Heidelberg: Springer, 2009, pp. 152–171. doi: 10.1007/978-3-642-00899-3_9.
- [55] D. Bera, K. M. van Hee, M. van Osch, and J. M. E. M. van der Werf, "A Component Framework where Port Compatibility Implies Weak Termination," in *Proceedings of the International Workshop on Petri Nets and Software Engineering, Newcastle upon Tyne, UK, June 20-21, 2011*, M. Duvigneau, D. Moldt, and K. Hiraishi, Eds., in *CEUR Workshop Proceedings*, vol. 723. CEUR-WS.org, 2011, pp. 152–166. Accessed: Feb. 08, 2026. [Online]. Available: <https://ceur-ws.org/Vol-723/paper11.pdf>
- [56] F. Pommereau, "SNAKES: A Flexible High-Level Petri Nets Library (Tool Paper)," in *Application and Theory of Petri Nets and Concurrency*, R. Devillers and A. Valmari, Eds., Cham: Springer International Publishing, 2015, pp. 254–265. doi: 10.1007/978-3-319-19488-2_13.
- [57] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," in *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, O. Nierstrasz, Ed., in *Lecture Notes in Computer Science*, vol. 707. Springer, 1993, pp. 406–431. doi: 10.1007/3-540-47910-4_21.
- [58] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 248–259. [Online]. Available: <https://ieeexplore.ieee.org/document/7851476>
- [59] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, in *SOCC '11*. New York, NY, USA: Association for Computing Machinery, Oct. 2011, pp. 1–14. doi: 10.1145/2038916.2038938.

ICT, Strategy & Policy

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl