TNO innovation for life

A method for analyzing performance on complex Cyber Physical Systems

PPS: From Methodology to Application in Industry

TNO Public ⟩ TNO 2023 R12562

20 December 2023

ICT, Strategy & Policy
www.tno.nl
+31 88 866 50 00
info@tno.nl

TNO 2023 R12562 – 20 December 2023

PPS: From Methodology to Application in Industry

A method for analyzing performance on complex Cyber Physical Systems

Author(s)	Konstantinos (Kostas) Triantafyllidis, Sobhan Niknam, Yuri Blankenstein
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	53 (excl. front and back cover)
Number of appendices	0
Sponsor	Sponsor Name
Programme name	Maestro
Project name	Maestro 2023
Project number	060.55514/01.01

The research is carried out as part of the Maestro program under the responsibility of TNO-ESI with ASML as the carrying industrial partner. The Maestro research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

Contents

1 Introduction

The advance in industrial equipment has resulted in the extensive development of cyber-physical systems (CPS) systems that integrate computational and physical capabilities [1]. Typical examples of cyber-physical systems are complex industrial equipment systems, i.e., semiconductor equipment, production printers, analytical instruments, etc. A key performance characteristic of such systems is their timing performance, typically expressed in throughput, latency, etc. In the cyber-physical system domain, the synergy between the cyber and the physical aspects of the system both contribute to its performance. High performance (regarding timing aspects) is typically a challenging tradeoff with other system qualities, like product quality, cost (operational, ownership, etc.), reliability, upgradability, security, etc.

In addition, cyber-physical systems continuously evolve through the years, providing additional functionalities, while pushing the boundaries in fulfilling both the functional and the non-functional (i.e., throughput) system requirements. This system evolution results in a continuous increase of the size and the number of implementation artifacts, i.e., code base. This, in combination with other evolution factors [2], i.e., legacy, make the artifact management in terms of customization, optimization and diagnosis a rather complicated task. Such management actions typically require in-depth multidisciplinary domain knowledge over multiple abstraction layers in both the computation and the physical domains.

To customize, diagnose and optimize cyber-physical systems there is an emergent need for techniques that (a) provide detailed and accurate system insight, which further acts as reliable input for (semi-)automated analysis techniques and (b) substantially reduce the need of having extensive domain knowledge. such techniques, methodologies, frameworks and industrial practices have been developed and used through the years, as discussed in [3]. The presented methods, adhere to the model-based performance engineering approaches that facilitate composition of systems with guaranteed performance by construction. Five focus areas are distinguished: (a) Performance architecting, (b) Model-driven design-space exploration, (c) Performance modeling and analysis, (d) Scheduling and supervisory control, (d) Data-driven analysis and design (including data collection and model learning).

The five focus areas are positioned on the V-model in Figure 1 (the 6th is the supporting to the methods tooling). In this report we propose the PPS framework [4] that starts by data collection and then making connections to model learning which is further used for state-of-the-art automated performance analysis techniques. As mentioned, elements of PPS can be utilized in multiple focus areas, though the starting point is the data collection area.

The initial industrial challenge that we try to address is the diagnosis and visualization of performance issues that limit a system's throughput. Starting from the data collection and by automating observations, mainly by capturing dependencies between abstraction layers and artifacts belonging to different development phases in models, PPS can be used in a reverse engineering manner and act as input for the other focus areas, i.e., provide input for performance modelling and optimization.

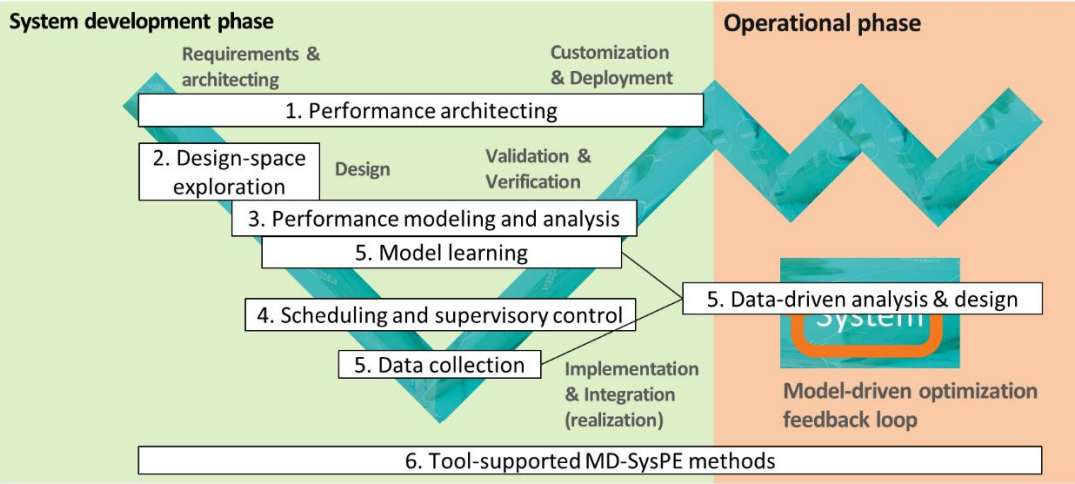

Figure 1. System development process with a positioning of the MD-SysPE focus areas.

This report focuses on the **fundamentals** of the PPS framework itself and its **application** on CPS. The fundamentals, namely the Lifecycle software architecture and its five stages, are first presented and discussed. While PPS has broad applicability for performance aspects **from design to optimization**, in this report the focus is on performance analysis techniques that address **issues occurring at system runtime**, i.e., Stimulated stage. The Stimulated stage is formalized by the TMSC modeling formalism and all performance analysis techniques adhere to the TMSC modeling formalism, thereby making PPS a general approach rather than tailored to specific system capabilities, architecture, or implementation. The domain specific concepts of each CPS are **modeled as extensions** to the generic models of the Lifecycle software architecture.

To verify the value on performance modeling and analysis, as well as to provide an example of how the PPS can be applied on a different CPS, the report provides a use case on a state-of-the-art complex CPS, the ASML TWINSCAN system on which the PPS is applied. The TWINSCAN system has a distributed component-based architecture. The communication between components and/or hosts is handled by an **enterprise middleware** that has similarities to the CORBA architecture framework. Due to the confidentiality of the original architecture implementation, the CORBA framework is used instead, to draw the mapping relations between the generic concepts of the PPS and the domain specific concepts of the TWINSCAN CPS. Finally, certain types of analyses are performed like critical path, slack and root-cause analysis for the identification of the root-cause of performance anomalies.

The following chapters discuss the PPS framework itself. Chapter 2 provides an overview of the PPS framework, and Chapter 3 describes the stages of the software lifecycle architecture which is the cornerstone of the PPS framework. Chapter 4 introduces the generic modeling formalism of the stimulated stage, the TMSC model. Chapters 2, 3 and 4 describe a generic abstract view of the PPS framework. As mentioned above, the PPS has been applied to a state-of-the-art and class-of-its-own cyber-physical distributed system with high throughput requirements. The architecture of the TWINSCAN is described in Chapter 5. Chapter 6 describes in detail the PPS phases, providing information in how PPS can be applied on CPS by building extensions of the generic PPS modeling formalisms. Chapter 7 presents the performance analysis techniques offered by the PPS and shows their application on the TWINSCAN application use case. Chapters 6 and 7 act as guideline and inspiration for the application of PPS in different CPSs and or domain(s).

2 Overview of the PPS framework

The main objective of the PPS framework is to provide adequate system execution insight over the system's execution time and enable performance analysis techniques aiming at facilitating (a) diagnosis of performance issues and identification of bottlenecks, and (b) system optimization for specific quality objectives, configurations, etc.

From a diagnosis perspective, the framework offers a means for both reactive and proactive diagnosis. The reactive diagnosis is meant for rapidly analyzing performance issues on realized/implemented systems (i.e., systems in the field) as well as prototypes, i.e., analyze the obtained traces after the system has been executed under a specific scenario, mode, configuration, etc. and identify the root-cause that led to a performance issue.

The proactive diagnosis aims at preventing performance issues at design- or implementation-time, before they occur in a realized system. Such type of diagnosis is typically performed using simulations or formal analysis techniques on models that replicate the system. The system execution models used for the diagnosis are also used during the system optimization process (a.k.a., design space exploration) for validating the performance of design alternatives, acting as a feedback loop.

In addition, the specification models of the system used in the proactive diagnosis are utilized for the construction of system design alternatives. PPS involves three phases, depicted in Figure 2: (a) Design and engineering, (b) Model forming, and (c) Diagnostics and visualization. These will be explained in the remainder of this chapter.

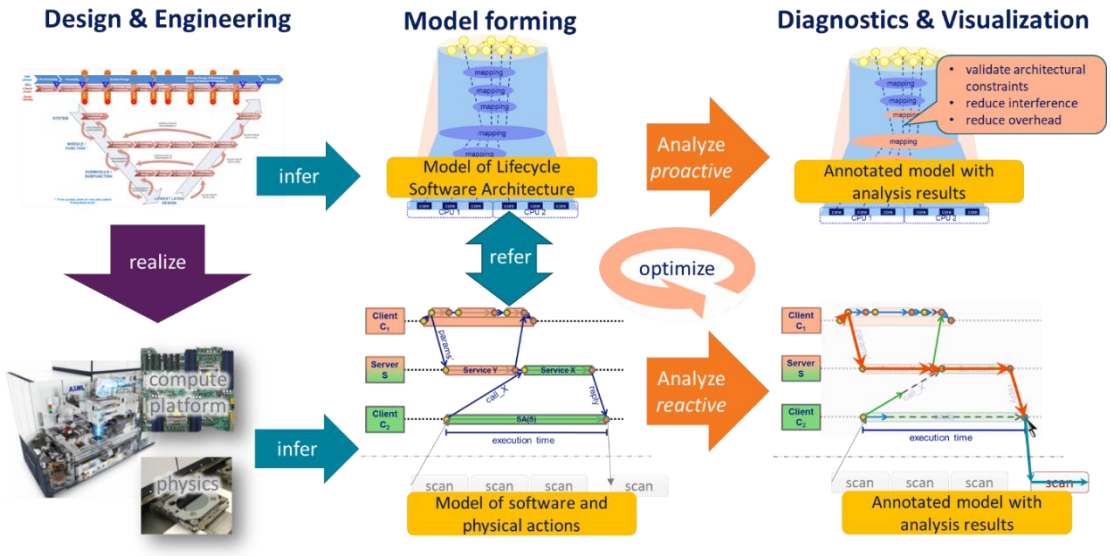

Figure 2. PPS reactive and proactive diagnosis framework.

The PPS framework is summarized as follows:
Our approach considers the software executions at two abstraction levels: the computation platform level (software, OS, computer hardware, etc.) and system actions at the physical layer (mechatronics, actuators, etc.). In order to diagnose the system when performance anomalies occur (i.e., execution

takes more time than normal), accurate logging at the right abstraction level and resolution is required. To achieve this, in the PPS framework, we introduce a probe set by instrumenting the source code at *strategic* places. The probe set aims at (a) capturing only the information that is required for the performance analysis while (b) limiting the intrusiveness of the tracing on the overall system performance. During system execution, the traces are generated and thus obtained for both software and physical actions.

The system traces are used as input in the model inference phase to reconstruct the model of execution. The inferred model depicts the flow and duration of the actions that have occurred during the execution of the system and can be typically used for the identification of performance anomalies. For the model of execution, we developed a generic formalism that captures the execution behavior (separate or combined) of software actions at the computation platform or physical layer. This formalism is the Timed Message Sequence Chart (TMSC), presented in [5]. This generic formalism enables decoupling of the modeling and analysis approach from domain concepts. Thus, the inferred TMSC models can be analyzed with generic performance analysis techniques.

However, the translation of domain-specific traces to domain-agnostic TMSC models requires domain-specific knowledge such that domain concepts expressed and encoded within the domain-specific traces can be mapped on the generic concepts of TMSC. The PPS framework captures the domain-specific knowledge in the form of a *lifecycle software architecture model*, which connects the software artifacts through the system lifecycle stages, i.e., from the specification to the runtime stage. The *lifecycle software architecture model* is inferred from existing artifacts in the product development process and encapsulates *mapping* and *transformation* relations between different artifacts in various lifecycle stages, such as the Specification, Implementation, Deployment, and Runtime stage (further distinguished in the Instantiated and Stimulated stage). An example of such a mapping relation is how an executed method observed at runtime relates to a specified interface, or how different instances of an executable are mapped on different processes, processing units or hosts. The inferred model of the lifecycle software architecture is not only employed to facilitate the mapping of encoded domain-specific concepts present in system traces to TMSC concepts. It is also used to create a reference/connection between generic modeling and analysis concepts with specific system artifacts, i.e., TMSC events and dependencies, to specific system artifacts, like components, interfaces, or operations. With this connection, performance analysis results can be linked back to specific system artifacts. For example, the root cause of a performance issue might be related to a specific deployment choice. Various generic analysis techniques like critical path analysis, root cause analysis and outlier identification analysis can be applied to the inferred TMSC model. The TMSC models, as well as the results of the analysis, are presented to the user using various visualizations focusing on facilitating the diagnosis at specific abstraction levels. Finally, a model of the lifecycle software architecture enables proactive diagnosis by analyzing mapping relations between artifacts and therefore being able to identify potential bottlenecks, i.e., when two artifacts “compete” for the same resource.

In the following chapters, we detail prerequisite knowledge, before further discussing the individual phases of the PPS framework:

1. Lifecycle Software Architecture
2. TMSC modeling formalism

3 Lifecycle Software Architecture

An important aspect of the PPS framework is the classification of artifacts and modeling elements in different stages during the software lifecycle. The stages that we consider are inspired by Koziolek et al. [6], where the author presents a view on the software component lifecycle. This study considers four levels of lifecycle stages: the *Specification*, *Implementation*, *Deployment* and *Runtime* level. While the study refers to the component lifecycle, our approach generalizes the concept to be applicable over all software infrastructure taking also into consideration the execution platform and the mappings of software elements on it.

The PPS lifecycle architecture considers an additional distinction in the *Runtime* level, which is divided in (Runtime-) Instantiated and (Runtime-) Stimulated stage. Next, each individual stage is described in more detail.

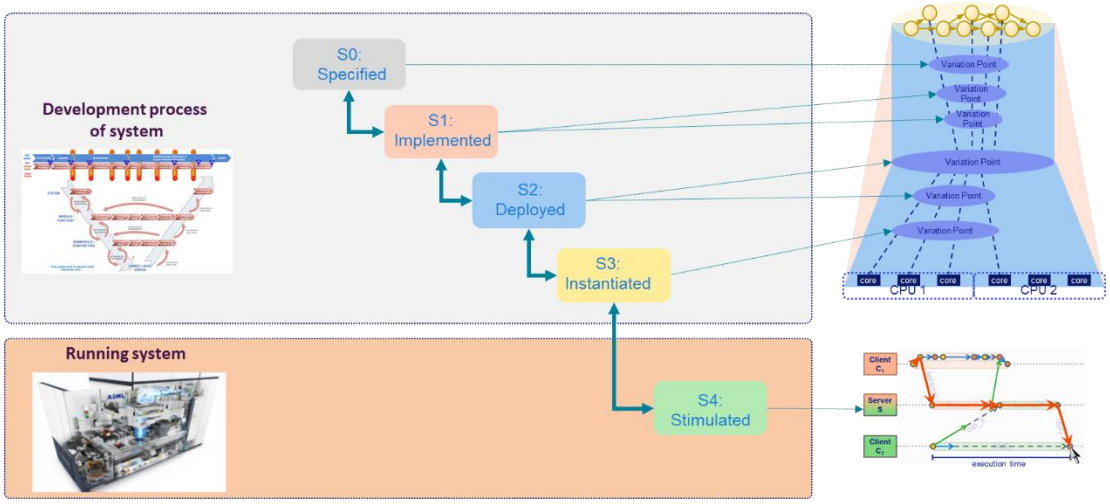

Figure 3. Lifecycle stages: Specified, Implemented, Deployed, Instantiated and Stimulated stage.

The first four stages, namely *Specified*, *Implemented*, *Deployed* and *Instantiated* contain modeling elements and artifacts from the system development process (i.e., the V-Model), as depicted in Figure 3.

a) The **Stimulated** stage considers the notion of time, (i.e., when the system is being executed) and its formal definition is captured by the TMSC modeling formalism. The stimulated stage may contain multiple executions of the software of the system. Each execution may differ based on the elements that are present, which may depend on the input/use case executed each time. At the right-hand side of Figure 3, the hourglass is modeling variation points a.k.a. the mapping relations between the artifacts living in each lifecycle stage. On top of the hourglass, the abstract view of the system function is positioned in the form of a graph. At the bottom of the hourglass, the execution platform is visualized, where the different software entities (runtime components) of the system are executed. Between the top (graph) and the bottom (execution platform) there exist artifacts belonging to different lifecycle stages from Specified to Instantiated stage.
b) The **Specified** stage contains artifacts like Component and Interface specifications. A component may contain multiple interfaces, an interface may contain multiple methods. Modeling at the Specified stage influences artifacts in the subsequent stages. For example, modeling an interface as singleton or multi-ton influences the number of instances of that interface at runtime. In the

Implemented stage, the components are implemented. Here, also many mapping relations are modeled between the specified component and interface and their implemented counterparts.

c) At the **Deployed** stage, the software is bundled in binaries and installed/deployed on a specific host.
d) At the **Instantiated** stage, the system is started, and the binaries are instantiated into runtime components, which are mapped on the operating system's specific runtime entities like thread, process, task. Those OS runtime entities are also mapped onto the hardware of the execution platform, i.e., CPU/Core, with specific scheduling policies dictated by a system configuration artifact or by the execution platform configuration.

In the transition from one lifecycle stage to another, there exist transformation processes. For instance, from the Implemented to Deployed stage, the build process takes place, resulting in the binaries that are deployed on a specific host.

Note: The meta-models provided in the subsequent sections are abstract representations of the artifacts living in the different lifecycle stages. The elements themselves, as well as the containment relations and their cardinalities, may differ depending on the system of interest. Our intention is not to fully cover the elements and processes present at the lifecycle stages and the transitions between the stages, but to provide meta-model examples that help the reader to better understand the lifecycle stages by positioning well-known artifacts on them. In addition, we do not strive to model all processes involved during a product development process, but the focus is only on processes involved in the PPS framework. For instance, many may consider that at specification stage the behavior of the interfaces is also defined, but since PPS framework does not utilize such models the specification stage does not model it as such.

3.1 Specified stage

In this first lifecycle stage, the specification of the software components takes place. Provided and required interfaces describe the contracts between the software components and encode the service specification. At this stage, the software interfaces are also specified by describing the methods/operations that each interface implements (abstract meta-model in Figure 4).

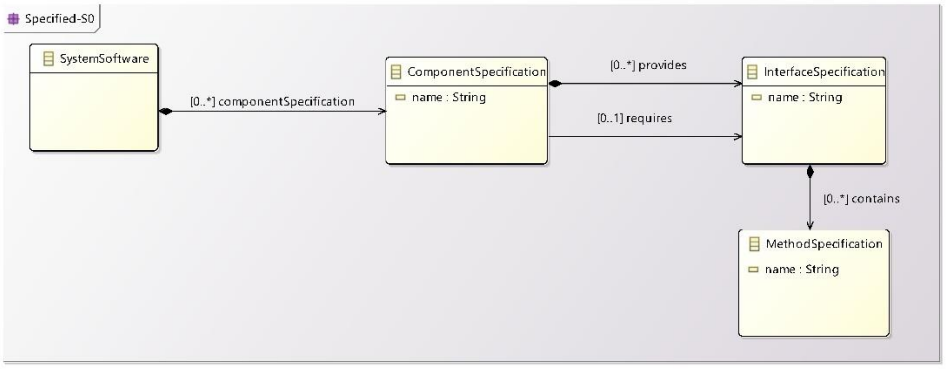

Figure 4. Abstract meta-model class diagram: visualizes conceptual modelling elements as artifacts that are part of the Specification lifecycle stage.

In this meta-model, the root element is the *SystemSoftware* that contains all software component specifications. The specifications of the components and of the interfaces may be part of a formal modelling language, which may generate implementation artifacts (i.e., IDL in CORBA). More artifacts may be present at this stage that specify other functional and non-functional (extra-functional) properties of components, interfaces, operations, etc. However, since this acts as an inspirational example, we leave to the reader to extend this meta-model with additional elements that are necessary in modelling a specific system (domain & implementation) for addressing a use case.

3.2 Implemented stage

In the Implemented lifecycle stage, the specified software components and interfaces are implemented. Depending on the software architecture, as well as the execution platform, a component may have more than one implementation, which may be related to a specific execution platform (i.e., programming language, type of processing unit, etc.) or other factors (i.e., quality of service, etc.). Figure 5 depicts modeling elements that may be part of the implemented stage, starting from the implemented component whose implementation in that particular example is bound to a specific target platform (i.e., CPU with specific instruction set, or programming language).

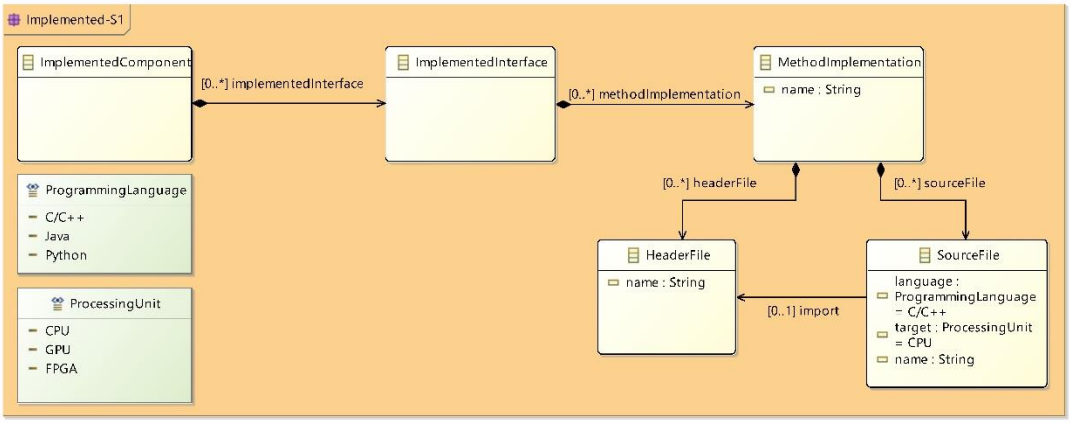

Figure 5. Abstract meta-model class diagram: visualizes conceptual modelling elements as artifacts that are part of the Implemented lifecycle stage.

3.3 Deployed stage

In the Deployed lifecycle stage, the implemented software components are built and bundled into binaries. At this stage, the binaries are placed in a filesystem from where they can be installed to the different hosts. The installation takes place during the transition from Deployed to Instantiated stage. During this transition the binaries are mapped (installed) to specific hosts. We consider, for the sake of simplicity, that the deployment of binaries onto hosts is static and performed before the system is initiated/started.

Figure 6 considers as binaries the executables and the libraries that are artifacts of the build process after the implemented stage. The binaries are then statically assigned/copied to specific execution host during the transition from Deployed to Instantiated Runtime stage.

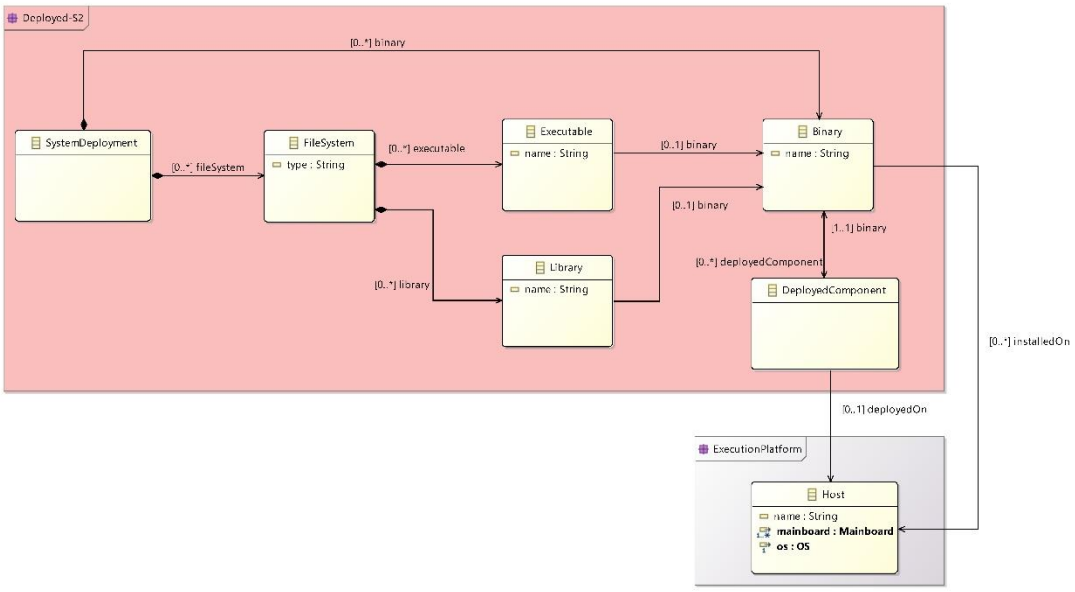

Figure 6. Abstract meta-model class diagram: visualizes conceptual modelling elements as artifacts that are part of the Deployed lifecycle stage.

3.4 Instantiated

In the Instantiated stage the system is started, and the components are instantiated and start living in memory. Here, the runtime component takes form and is mapped on OS execution entities, i.e., on a specific thread that is mapped on a specific process, etc. Communication ports are created and instantiated, enabling communication among the runtime components. The OS execution entities, i.e., processes and threads are mapped on specific processing unit, with specific scheduling policies, priorities, etc. In the Instantiated stage, we consider domain-agnostic and domain-specific modeling elements (see Section 4.2.1).

3.5 Stimulated

The Stimulated stage is a specialization of the runtime instantiated stage. Here, the initialized system starts the functional work by executing the software with specific data input. The observed behavior of the software can be modeled in a graph which contains executions with timestamps and dependencies between them. A specialization of such a graph is the Timed Message Sequence Chart (TMSC) formalism, which is the cornerstone of the analysis techniques considered in PPS, and is discussed in Chapter 4.

Note: Depending on the domain and the system/software architecture, the distinction between lifecycle stages may differ. For instance, the specification, implementation and deployment could be merged in a single stage, depending on the way of development, i.e., model-based engineering could be employed for specification and automated generation of implementation and build artifacts.
In addition, for automated inference of TMSC models, it is not required to have all stages and elements in place. The completeness of the models depends on various aspects and on the use case that the PPS methodology will be applied on.

Note: The transitions from one lifecycle stage to another stage are considered as a transformation process that transforms and connects artifacts from one stage to the other stages. For instance, from

the Implemented to the Deployed stage, during the transformation, the build process takes place. From Deployed to Instantiated, the binaries are copied/installed to specific hosts, etc.

4 TMSC model

The cornerstone of the modeling and analysis techniques in the PPS framework is the TMSC formalism. TMSC model is the main modeling aspect that is used for capturing the Stimulated stage of the lifecycle software architecture. In Section 4.1, we discuss how the TMSC modeling formalism was inspired by Message Sequence Chart (MSC) models. In Section 4.2, we present the formalization of TMSC as presented in [5] and a "naïve" TMSC meta-model derived from it. Section 4.3 presents the refined and complete TMSC meta-model that is used in PPS for modeling and performance analysis of models in the Stimulated stage.

4.1 Inspiration

The TMSC model is a derivative of UML Sequence Diagrams [7], combined with the aspect of time of events. A sequence diagram is described as a diagram that depicts interaction and focuses on the message interchange between several lifelines. As depicted in Figure 7, a TMSC model is an MSC, rotated by 90 degrees, where the lifelines are positioned horizontally, forming so-called swim-lanes. Each swim-lane represents an autonomous execution entity, following the principle "Run to completion", imposing sequential execution per lifeline. The overlapping executions of the MSC on the same lifeline are shown as stacked boxes in TMSC (equivalent to a call stack). In addition, in TMSC the execution labels (names) are drawn within the boxes. The arrows in TMSC represent asynchronous messages, i.e., the sender executor continues its execution directly. The horizontal axis shows the actual time. The length of an execution represents its duration. The projected distance of the start and end events of an arrow on the horizontal axis denotes the latency of the message passing. Important to be mentioned is that fragments, such as loops, alternatives and references, are not supported in TMSC.

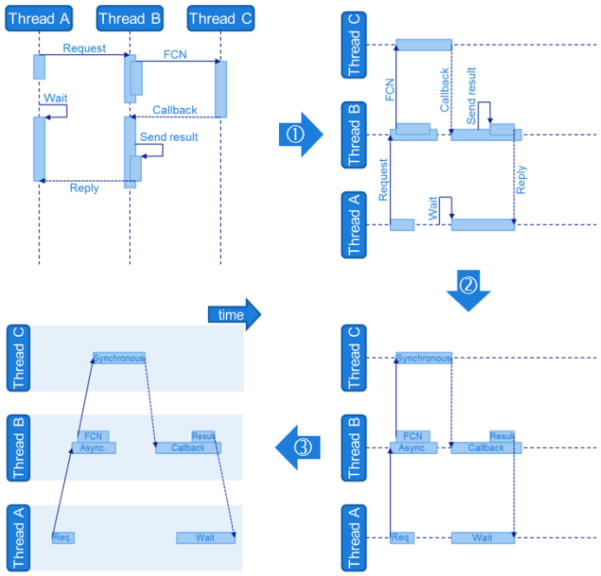

Figure 7. From Message Sequence Chart (MSC) to Timed Message Sequence Chart (TMSC). Each lifeline is a thread, each action is a function call. The message exchanges depict the communication between the functions. The message exchanges are annotated with the type of the message exchange, i.e., Request, FCN, Callback, Reply, etc. In the first transformation the MSC is rotated by 90 degrees. In the second the properties of the message exchanges become part of the actions (function calls). In the third transformation the notion of time is added, and therefore the message exchange arrows become sloped; the start of the arrow depicts the moment the message starts its execution on the caller thread and the end of the arrow encodes the moment the message has completed its transfer.

4.2 Formalization

The introduction and formalization of the Timed Message Sequence Chart (TMSC) is discussed in [5]. A TMSC is formally defined in [5] as, $TMSC = (E, \rightarrow, m)$, where E is a set of events, $\rightarrow$ defines a timing constraint relation between events and m is a partial function that maps the timing constraint relations to messages. We "borrow" the informal description and example of a TMSC from [5]:

"*The example presented in* Figure 8*, shows a simplification of the use case that will be discussed in later section. The picture shows on the vertical axis components C_1, C_2 and C_3, each of which executes functions and exchanges messages at different moments in time. The horizontal axis on the bottom of the figure displays the time during which functions are executed and at which times messages are sent and received. With every function execution we associate a start event, shown as an open circle and a finish event, shown as a closed circle. Events occurring on the same component are totally ordered, whereas events across components are partially ordered according to the messagea exchanges between components. Start and finish events have unique labels, but for reasons of brevity these labels are omitted from the picture. Function executions are shown as rectangles labeled with the names of the functions that are executed. For instance, on component C_1 function l is executed, starting at time 0 and finishing at time 19. Functions can be executed within the context of another function representing nested local function calls. In the figure, nesting is represented by positioning the function execution being invoked on top of the function that invokes it. In the figure, the execution of function l involves repeated executions of functions f_1, f_2, f_3 and f_4. A function that is nested in another function must always start no sooner and finish no later than the function it is nested in. This ensures that the call stack is well-formed. The 'bottom' of the call stack represents the function that is executed on behalf of another component. Such an execution must be finished before another call can be serviced. Hence, components can only serve one remote function call at a time.*

Components can exchange messages. These are shown as directed arrows labeled with a message name. For instance, upon the start of function f_1, component C_1 sends message g_1^{call} to C_2, which signifies the start of the execution of function g_1. After completion of this execution, component C_2 returns reply message g_1^{reply} to C_1. This communication pattern encodes a remote function call. Arrows representing message exchanges always leave from (start or finish) events and terminate at (start or finish) events. Exchanging a message takes some amount of time. This time is indicated as a label of the message exchange. For instance, the leftmost message exchange between the start events of f_1 and g_1 is labelled 0.1. This indicates that the start of g_1 cannot occur earlier than the start time of f_1 plus 0.1 time units. In the figure, g_1 starts precisely 0.1 time units after the start of f_1, but in general this need not be the case. Thus, the time label encodes a lower bound timing constraint.

Figure 7 *shows three different types of remote function calls that are supported by many component-based software frameworks. In this paper, we only discuss these patterns, albeit that in practice other patterns also exist. The first type is exemplified by message h_1^{trig} between components C_2 and C_3. This type of remote function call is called a trigger. The sender of such a trigger does not expect a reply and can thus continue its execution. The second type is a blocking call, an example of which is shown by messages h_2^{call} and h_2^{reply} between C_2 and C_3. A blocking call starts with a message between the start events of both functions and ends with a message between the end events running in the opposite direction. Here, the caller is waiting idly for a response from the callee. The third type is a non-blocking call, exemplified earlier as message g_1^{call} followed by message g_1^{reply}. Upon calling function g_1, C_1 continues executing functions f_2 and f_3, where f_3 is a call that idly waits until the reply g_1^{repl} from C_2 has been received. Notice that TMSCs do not support explicit concepts to express function call patterns."*

Examples of additional call patterns that enable concurrency using the TMSC modeling formalism are presented later in this document, when an explicit use case is considered and are visualized in Figure 16 in Section 6.1.2.

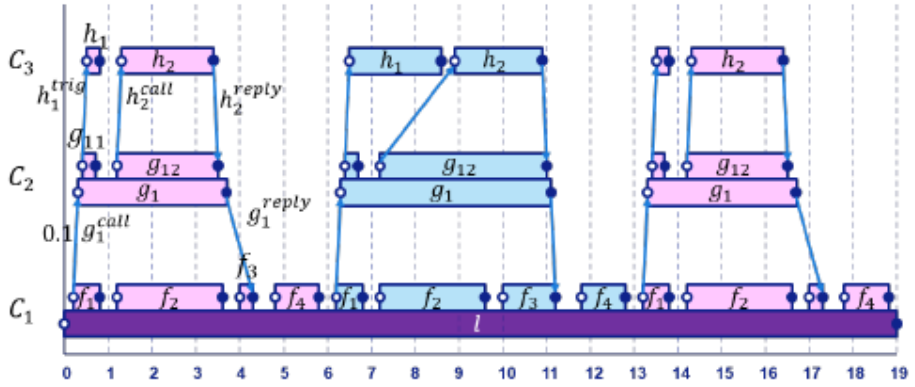

Figure 8. A schematic view of a timed message sequence chart.

The meta-model that adheres to the formal definition of the TMSC is depicted in Figure 9. From the definitions, we observe that a TMSC model contains entities from both specification and runtime lifecycle stages. The specification entities are statically defined artifacts, which typically encode domain aspects; their presence facilitates the cross-reference between runtime entities to domain aspects. Such specification entity is the *Function* model-element. The runtime aspects are domain agnostic, and their existence starts when system is started. The runtime stage is further distinguished in *Instantiated* and *Stimulated* stage (see Sections 3.4 and 3.5). The former, contains elements that do not have the notion of time, i.e., *Component*, while the latter contains elements which are distinguished by the notion of time (timestamp), i.e., *Event*, *Dependency*, *Execution*. The TMSC meta-model also contains elements that are bound to facilitate performance analysis, i.e., storing analysis results.

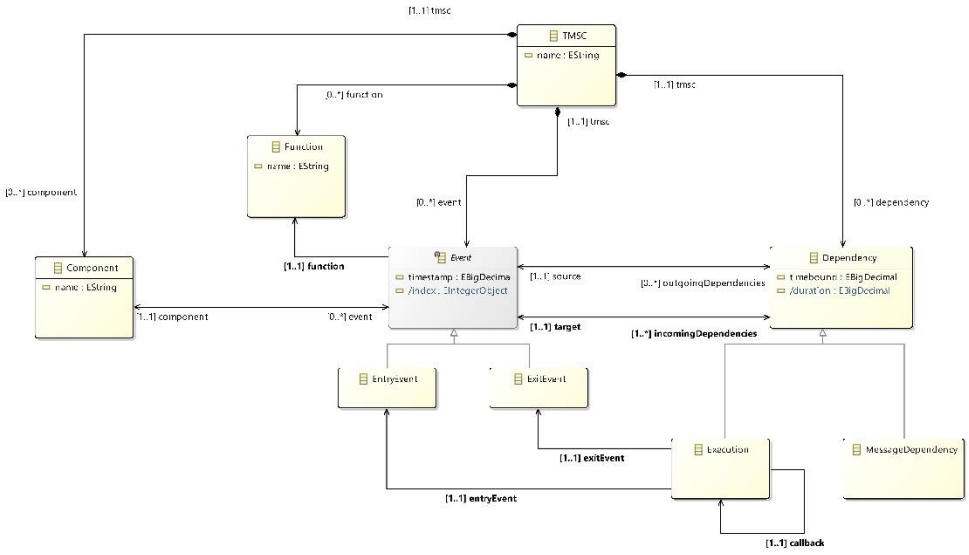

Figure 9. Formal TMSC model.

Domain-agnostic elements
A **TMSC** consists of entry and exit events connected via dependencies. Those events are forming *Executions* which are mapped to their executor (i.e., *Component*).

Domain-specific elements
The consideration of elements living in Specification stage enables the association between events and specific *Functions* and *Interfaces*. The latter is crucial for diagnostics, where for an analysis, an

identification of an issue should be coupled to specific domain aspects, i.e., relate event executions to functions that the system is executing.

From the definitions, the example, and the meta-model, it is apparent that the *Component* modeling element as used, includes in an abstract sense the lifeline and executor concepts. In reality, when addressing real-world cases, those concepts are separate and exist on different lifecycle stages. The executor in principle exists in the runtime instantiated stage (i.e., runtime component bound to thread), thereby pre-existing the TMSC model instance(s). For this and other practical/implementation reasons, the derived TMSC meta-model from the formal definition in [5] has been refined to facilitate transformation of domain-specific traces to generic TMSC models and is discussed in combination with the modeling of the lifecycle software architecture elements.

4.3 Refined TMSC meta-model

The model inference translates the obtained traces to models that adhere to the *refined* TMSC meta-model that is presented in Figure 10. Next, the additional elements compared to the formal TMSC meta-model, as presented in the previous section, are discussed.

A ***TMSC*** is a directed acyclic graph with events and dependencies, and some additional structural properties. Two specializations are considered the *FullScopeTMSC* and *ScopedTMSC*.

A ***FullScopeTMSC*** contains all events and dependencies of the TMSC model. It has the *startTime* and *endTime* attributes which are timestamps derived from the TMSC model, denoting the first and last event, respectively.

A ***ScopedTMSC*** is a sub-graph of the whole TMSC model (full-scope). It is defined for facilitating decomposition of the TMSC model into sub-graphs (indication of a path), i.e., collections of events and dependencies that have specific properties, part of the same activities, etc. Each *ScopedTMSC* can be considered as part of the parent TMSC model and allows application of analysis techniques, i.e., critical path analysis, locally in a sub-graph. The dependencies of a child scope are contained by its parent scope.

All domain-specific and domain-agnostic elements are modeled as generalizations of the modeling element ***PropertiesContainer.*** This facilitates the seamless extension of specific modeling elements, i.e., events with additional attributes, that are required for a specific type of analysis, e.g., slack analysis. Though this allows rapid prototyping or customer-specific annotations, for every property that is added to the model one should ask whether this property is a primary citizen of all models. If so, the property should be promoted to an attribute in the meta-model.

Lifeline is the modeling element as defined in the UML Message Sequence Chart formalism [7]. The lifeline is equivalent only to the instantiated nature (viewpoint) of the Component modeling element as defined in the abstract TMSC meta-model. A lifeline contains the behavior of an Executor, in the form of a sequence of events, over a specific time span.

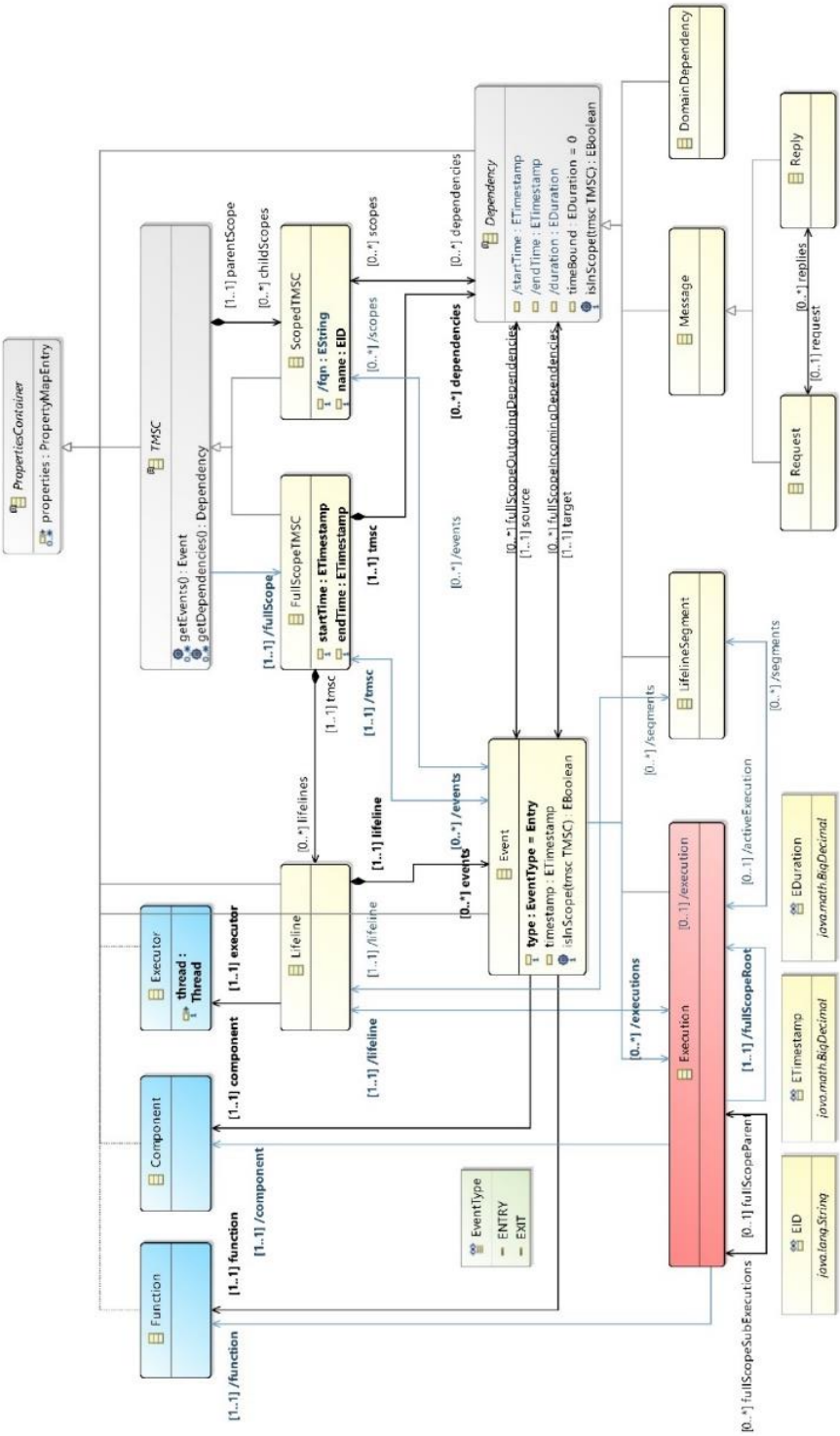

Figure 10. TMSC meta-model in Ecore modelling semantics.

5 Application domain – System of interest

From this moment and for the rest of this report, we focus on particular aspects of the PPS framework by applying and evaluating it on a state-of-the-art CPS, a lithography scanner. Lithography scanners, such as the TWINSCAN system in Figure 11, are highly complex cyber-physical systems used in the manufacturing of integrated circuits [5] [8] [9]. The system is composed of various subsystems which are orchestrated to achieve optimal **throughput**, i.e., the number of processed wafers per hour.

As discussed in Chapter 1 throughput is one of the determining key performance indicators of high-tech systems. A lot of attention is given to the optimization of such complex systems to achieve the highest possible throughput. A key architectural principle that is often applied, considering the cost of developing the physical elements of cyber-physical systems, is: “Throughput should be determined (limited) only by physics”. This means that other (i.e., software) actions should always be executed in the shadow of the physical actions, staying away from the critical path that determines the throughput.

Figure 11. A TWINSCAN system and its subsystems.

In addition to system throughput, system accuracy is another key performance indicator. To drive this performance indicator, an increasing number of software actions in the form of correction algorithms, models, and control loops are added to these systems. Even with these extra computations, these actions have to be kept out of the critical path. For cost reasons, these actions are executed on shared platform resources, making their timing execution variable and unpredictable. The combination of (a) the varying customer operating requirements, (b) the ever-increasing number of software actions, (c) the decreasing timing budgets and (d) the increasing multiplexing on platform resources, puts an increasing amount of stress on the key architectural principle, i.e., it is ever more challenging to keep the software out of the critical path [8].

5.1 Software platform – CORBA architecture

The software platform of the TWINSCAN system can be considered an enterprise implementation of the CORBA architecture. CORBA, which stands for Common Object Request Broker Architecture, is a standard, defined by the Object Management Group (OMG) [10], and designed to facilitate transparent

communication of systems that are deployed on different hosts, while enabling interoperability [11] between software (and its implementation(s) in different programming languages) and execution platforms for instance different operating system, different computing ana execution platform, etc.).

The **Interface Definition Language** (IDL) provides the primary way of describing interfaces in CORBA, i.e., data types, methods and their signatures. IDL is independent of any programming language. Mappings, or bindings, from IDL to specific programming languages are defined and standardized as part of the CORBA specification.

The central CORBA functions, services, and facilities, such as the **Object Request Broker** (ORB) and the **Naming Service** (discussed in following paragraphs), are also specified in IDL.

Figure 12 depicts the basic elements of the CORBA architecture. These are the IDL stubs from the client side, the Object adapter, the Skeleton interface from the server side and the ORB that binds client and server sides. Each of those elements and their contribution in the client/server paradigm are further explained in more detail in the following paragraphs.

The **IDL Stub** consists of functions generated by the IDL interface definitions [12]. The functions are a mapping between the client and the ORB implementation. A function within an IDL stub is called by the client just as if it was a local function at the client side.

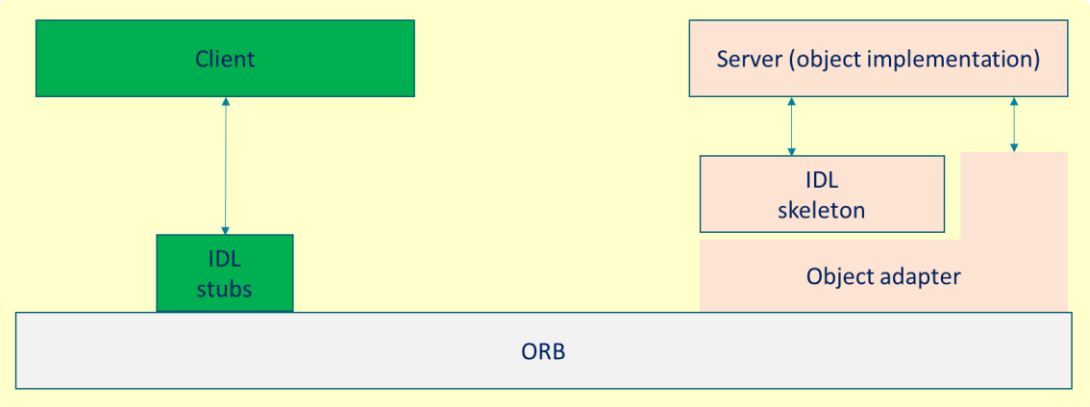

Figure 12. CORBA architecture

The **Object Request Broker** (ORB) provides a communication hub for all objects. It uses a broker to handle messages between clients and servers, providing object location transparency.

Object clients and servers make requests through their ORB interfaces. The client ORB provides a stub for a remote object. Requests made on the stub are transferred from the client to the server that contains the implementation of the target object. The request is passed on to the implementation through the object adapter and the object's skeleton interface.
An **object adapter** is the primary means for an object implementation to access the ORB services, such as object reference generation. It acts as the glue between CORBA objects and IDL interfaces. An object adapter is responsible for creating remote object references for CORBA objects. The object adapter provides a general facility that "plugs" a server object into a particular CORBA runtime environment. All server objects can use the object adapter to interact with the core functionality of the ORB, and the ORB in turn uses the object adapter to pass along client requests and lifecycle notifications to the server object.

The **Skeleton interface** is specific to the type of object that is exported remotely through CORBA. Its main contribution is to provide a wrapper interface that the ORB and object adapter can use to invoke methods on behalf of the client or as part of the lifecycle management of the object.

Typically, an IDL compiler is used to generate the IDL stubs and skeleton interfaces for a particular IDL interface; this generated skeleton interface will include calls to the object adapter that are supported by the CORBA environment in use.

5.2 Lifecycle software architecture

This section provides an example of a minimum viable lifecycle software architecture that describes the software and the execution platform architecture that was presented in the previous section. The lifecycle stages included in the lifecycle software architecture, as well as the modelling aspects and elements that are considered, depend not only on the system aspects (i.e., architecture), but also on the use cases that the lifecycle software architecture will be used for. In this case, the primary goal is not to be complete, but to demonstrate the application of the PPS framework on a specific use case by providing an inspirational example that can be easily extended and applied to other systems.

For simplicity and use case specific reasons, we consider three lifecycle stages: Specified, Implemented and Instantiated, modelled in separate interconnected metamodels. The Deployed stage is not present here, since its elements will not be of any use on the use case/example to which this lifecycle software architecture will be applied.

Note: A few of the modelling elements are greyed out. Those are the elements that will not be actively used in the upcoming TMSC modelling and analysis use cases. The reason they are included in the meta-model is to help positioning the CORBA architecture aspects, as explained in Section 5.1, with generic aspects positioned at the Specified, Implemented, and Instantiated stages (i.e., model of server and client side).

5.2.1 Specified

In the Specified stage, there exist three major modelling aspects. The meta-model visualized in Figure 13 captures the relation between component and interface via the provided and required relationship. In addition, each interface is composed of a few operations. Considering CORBA, the specification of an interface is done in the IDL modeling element. Each specified interface may be composed of several operations that are exposed/provided to the external world in the form of services. From the IDL, the *IDL Stubs* for the client and *Skeleton Interfaces* for the server (service implementation side) are generated, placed in the Implemented stage and used by the client and server sides, respectively.

5.2.2 Implemented

In the Implemented stage, the function is the implementation of the operation (Specified stage) and is offered at the server side and called at the client side. At the client side (see Section 5.1), a function is called via the *IDL Stubs* and at the server side the function call is bound to the actual function's implementation via the *Skeleton Interface*. For that specific reason, a function has two natures, the caller (client) or *IPCClientFunction* and the callee (server) or *IPCServerFunction*, as modeled in Figure 13. In addition, we choose to model the *IPCFunction* as a specialization of the function modeling element, since there could be internal functions at implementation level that do not follow the CORBA principles (i.e., inter-process communication), but are still visible in the Stimulated stage.

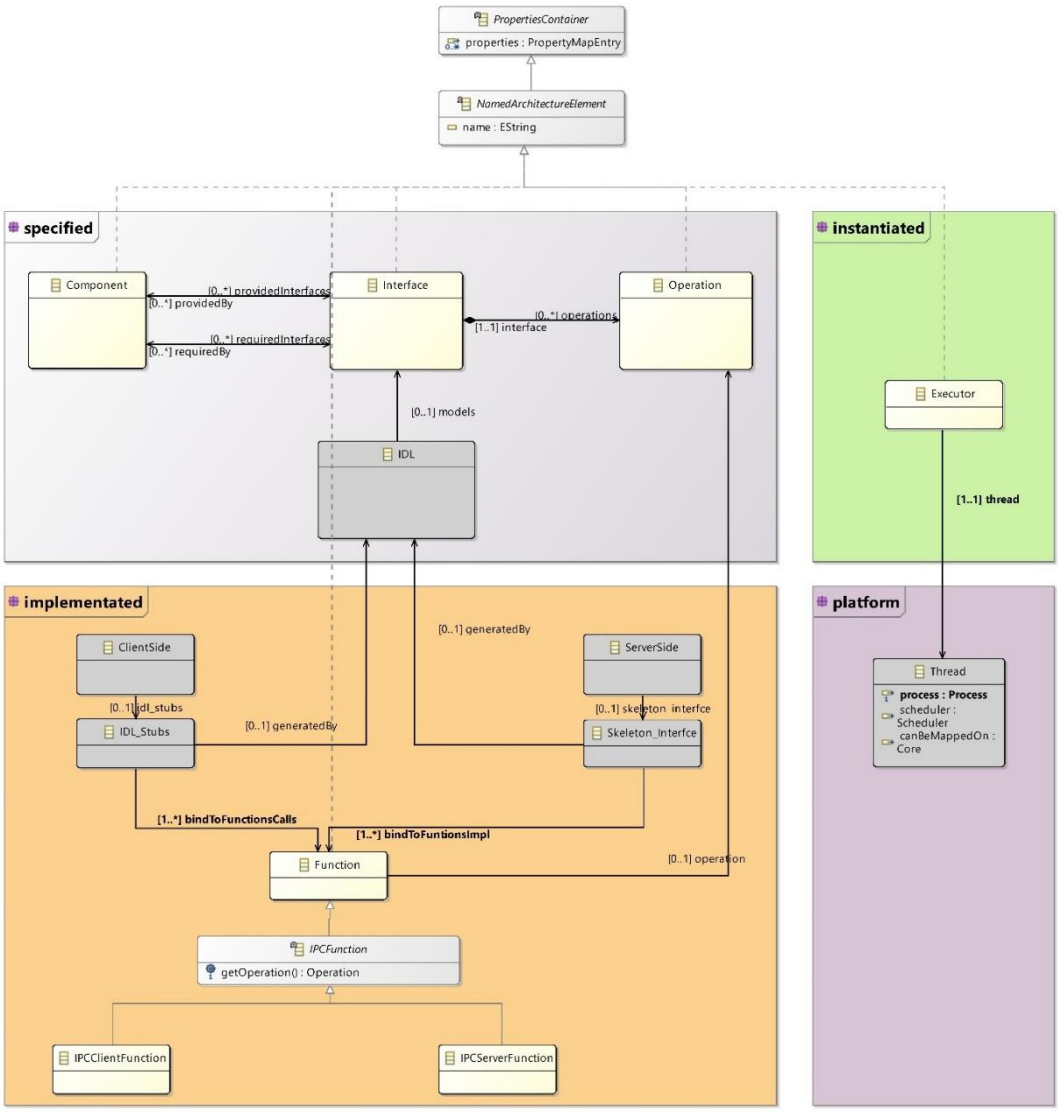

Figure 13. Lifecycle software architecture for CORBA.

5.2.3 Instantiated

In the Instantiated stage, the binaries resulted after the build and deployment processes (at Deployed stage) are instantiated and their instances mapped onto Executors. Executor is the modelling element where a lifeline of a TMSC is executed. The Executor itself is a domain-agnostic modelling element. Its mapping to a domain-specific element is performed by mapping the Executor element onto a specific platform aspect. In this particular case, there is a one-to-one mapping with a thread.

The execution platform meta-model is also modeled. In this case, we consider a multi-host Unix-based system, consisting of the software platform (OS), hardware platform (i.e., CPU, memory, hard disk), and the mappings between software and hardware elements. The meta-model visualized in Figure 14 is not complete (i.e., network hardware entities are missing), but it provides an overview of important concepts, such as process/thread and how they map on processing unit(s).

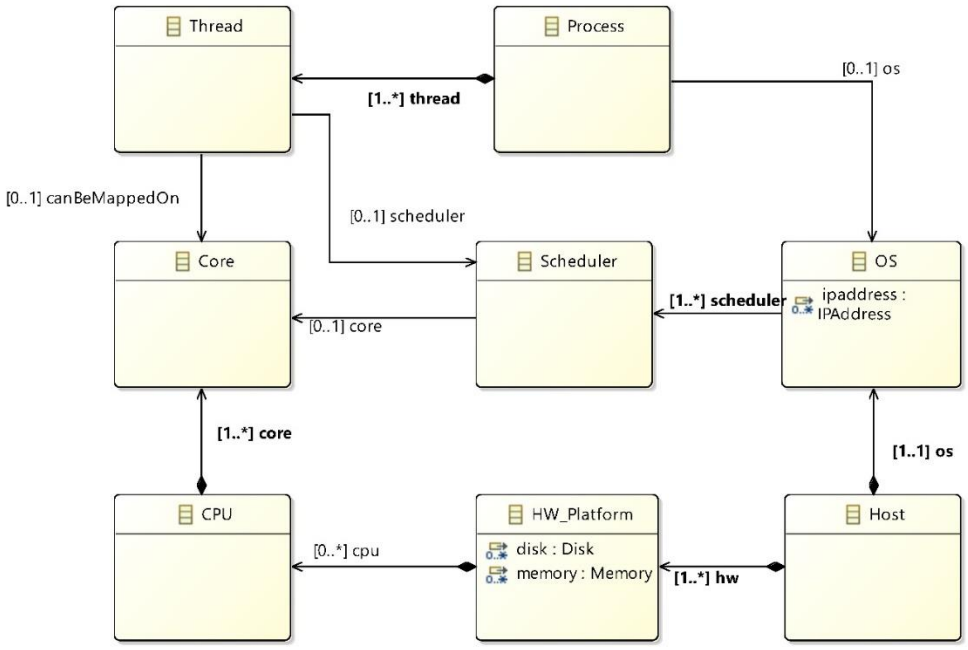

Figure 14. Platform meta-model.

6 PPS phases applied to CORBA use case

As mentioned in Chapter 2, the PPS framework considers three phases: 1) Design & Engineering, 2) Model Forming, and 3) Diagnostics and Visualization. The following sections present in more detail the actions that have to be taken in order to apply the PPS framework on a use case. Throughout the phases, we consider the system that has been already presented in Chapter 5.

6.1 Model inference

In order to automatically infer a model of the execution of the system, consistent tracing is a prerequisite. In addition, the tracing framework should be non- or at least minimally intrusive in terms of system performance.

This section discusses what is the essential information that should be traced, such that we can obtain TMSC models, and provides additional information on how to do so. The instrumentation approach (how to…) highly depends on the system of interest, i.e., software and execution platform, timing resolution, etc. Explicit examples are given based on this use case, but this approach may need adaptation when the software and execution platform of the system of interest is different.

6.1.1 What should be captured in traces

For automated inference of TMSC models, sufficient logging should be in place for the generation of traces that contain the required information.

As discussed in Section 4.2, the domain-agnostic part of a TMSC model is composed of **events** (entry and exit) characterized by **unique timestamps** and **relations** between the events, which are used to reconstruct the events' dependencies. That part of the TMSC is general, but in a domain-specific context, this generic modeling part needs to be coupled to specific domain elements, such as events coupled to **functions**, functions coupled to **interfaces** and then mapped on **executor(s)**.

Note: All the above-mentioned elements are essential to successfully reconstruct a TMSC model from traces. For this use case which acts as example, we set up a probe mechanism that provides all required information/logging for the reconstruction of the Stimulated Stage (i.e., TMSC). To realize the relation between the Stimulated stage and the other lifecycle stages, the combination of other traces/logs may be needed. For instance, during system initialization logs are collected, which represent the status at Instantiated stage, i.e., which process is instantiated on what host. The information of those logs may be essential for connecting Instantiated with Stimulated stage. Information such as process and thread IDs (essential for mapping relations) cannot be retrieved by design artifacts due to their dynamic assignment by the execution platform.

Traces may not be the only source for TMSC model inference. For example, dependencies between events that cannot be extracted from existing traces, may be possible to be extracted by domain knowledge injection (i.e., functional behavior, communication implementation, etc.).

6.1.2 Setting up probes

In order to obtain a trace that contains both domain-agnostic and domain-specific information that is essential for the Stimulated Stage (TMSC model) of the Lifecycle software architecture, we set up probes to obtain the required information in a trace format. Let us consider a client/server case, where the caller at the client side calls a function in a remote blocking fashion at the server side. To be able to reconstruct that call, there should be probes at multiple places both at the client and the server side. Abstracting from the client/server communication and implementation mechanism and focusing only on the functional part, to reconstruct such a call, multiple timestamps should be captured: (a) when the call is triggered at client side, (b) when the call started to be handled at server side, (c) when the call handing has finished at the server side and the reply is sent to the client side and (d) when the reply has reached the client side. The number of probes that has to be set up increases when taking platform aspects, such as the communication mechanism, OS, and hardware aspects, into account.

As already discussed in Chapter 5, the system we consider is based on CORBA. There is transparency between the caller and the callee, meaning that they may be executed on the same or a different host, in the same or a different process or thread. The ORB plays a key central role in this transparency, facilitating the identification of the callee that hosts the function implementation, the transfer of the function call to the callee and the translation/transportation of the results from the callee to the caller. This central role opens opportunities to globally reconstruct the timing behavior of the software execution by probing and analyzing function calls passing through the ORB middleware components.

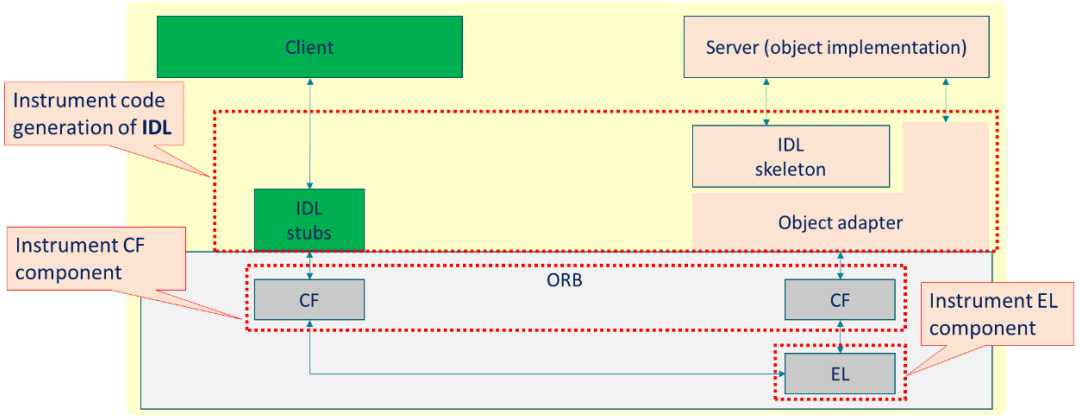

Figure 15. Probe setup in strategic places.

In the case of the TWINSCAN system, as depicted in Figure 15, the ORB infrastructure also encapsulates the Communication Framework (CF), which is an enterprise solution on top of TCP/IP. In addition, an Event Loop (EL) enterprise solution is integrated in ORB. This event loop scans the file descriptors for new events being present and then schedules them in the queue to be handled from the server.

Event Loop: The main functionality of the event loop is to detect and handle events. The implemented event loop is able to handle a single event at a time. An event handler (callback function) is registered for each event and is executed when the event loop detects the corresponding event. The handler processes the event and can raise other events for itself or for other (remote) processes. The event loop is blocked during an event handling. If events arrive while the event loop is blocked, they are placed in the queue until the event loop is released. The queue in the current implementation considers priority-based scheduling combined with an underlying FIFO policy.

Communication Framework: The communication framework (CF) is a message passing mechanism between clients and servers. CF uses socket communication as transport mechanism between clients and servers. Local and remote communication is based on the TCP/IP protocol. CF registers callback functions on EL and when a message arrives in EL, the appropriate message handler is called to execute the request.

Note: The ORB middleware components are part of each host in a distributed system. In distributed systems, each host maintains its local clock. It has been observed that the clocks may not be fully synchronized, especially when the system is running for a short time period, thereby imposing additional challenges during the TMSC model inference. Many fundamental solutions may apply depending on the system architecture and implementation (i.e., global clock), as well as heuristic-based techniques which relate execution events from different hosts. The present use case does not consider those challenges and considers clocks to be sufficiently synchronized.

Communication patterns

The enterprise CORBA implementation supports three major types of remote procedure calls from the client perspective: the function, the trigger and event call type. The function calls are further distinguished in blocking, request/wait and non-blocking (CORBA-AMI callback) [13]. The functions calls can be handled by the server in a synchronous or asynchronous manner.
In detail the different types of calls are:

a. **Blocking:** The client requests a function call and waits (cannot perform other execution tasks) till the reply has arrived from the server (see Figure 16-a). The server handles the request (synchronously or asynchronously) and sends the reply to the client.
b. **Request-Wait:** The client requests a function call and the difference with the blocking call is that it can perform other actions before it starts waiting for the reply from the server (see Figure 16-b). At the server side, similarly to the blocking call, the request can be handled synchronously or asynchronously, and a reply is sent to the client.
c. **Non-blocking (NBC):** The client registers a callback handler and sends a request to the server. The client waits asynchronously, meaning that it can continue with executions of other tasks (see Figure 16-c). The server processes the request (synchronously) and sends the reply to the client. By the moment the client receives the reply, it executes the callback function.
d. **Raise trigger:** The client sends a request to the server, but no response is expected and therefore the client can continue with the execution of other tasks. The server processes the trigger (see Figure 16-d).
e. **Subscribe to events:** The client registers a callback and sends a subscribe request to the server. Similarly, to raise trigger, the client does not block waiting for response. The server handles the subscribe request asynchronously (see Figure 16-e). As soon as the specific event is raised on the server side, the server replies to the subscribed clients (more than one client may be subscribed for a particular event). The client handles the received event and subsequently executes the callback function.
f. **Unsubscribe from events:** The client sends an unsubscribe event request to the server and the server handles the request (see bottom of Figure 16-e).

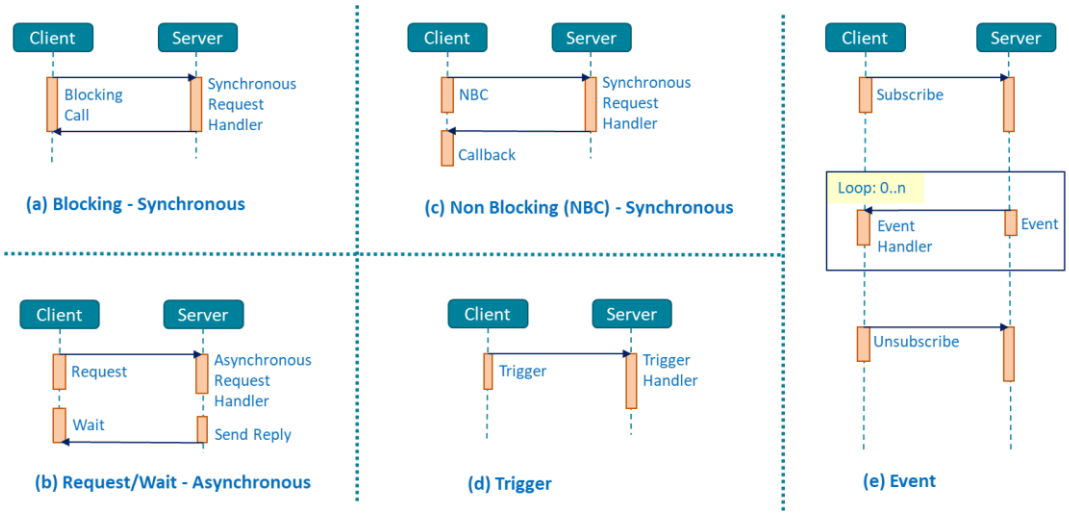

Figure 16. Supported remote procedure calls.

What needs to be traced

Let us assume a blocking remote procedure call from a client that is handled on the server side synchronously (Figure 16-a). The execution of such a call, including the CF (Communication Framework) and EL (Event Loop) infrastructure, results in the TMSC model depicted in Figure 17. At the client side, a call that is part of the IDL interface is triggered via the IDL stubs (dependency 1 in Figure 17). At this stage, the stubs connect to the CF in order to pass the message to the server. Via the “Write” function (part of the CF interface) the message (call with arguments) is passed to the server side (dependency 2 in Figure 17) and placed in a queue (file descriptor) waiting for the EL handler to pick it up. The EL handler handles the event (dependency 3 in Figure 17), and the CF component via the “Read” function starts reading and unpacks the message (dependency 4 in Figure 17) and then connects via the object adaptor and the IDL skeleton to the call (dependency 5 in Figure 17). As soon as the call is handled and completed its execution, the result is transmitted back to the client side via the CF “Write” function (dependencies 6, 7 in Figure 17). At the client side, the result is read and returned to the initial blocking call (dependency 8 in Figure 17).

One may consider different tracing strategies. Tracing only the functional aspects of the system (i.e., IDL interfaces), the path that is visualized as dotted arrows in Figure 17, namely the start and end of the *Call* at the client side and the start and end of handling the Call (*Call-Handler*) at the server side, enables decoupling between functional aspects and infrastructure design/implementation (i.e., implementation of the communication (CF) and event loop (EL) framework).

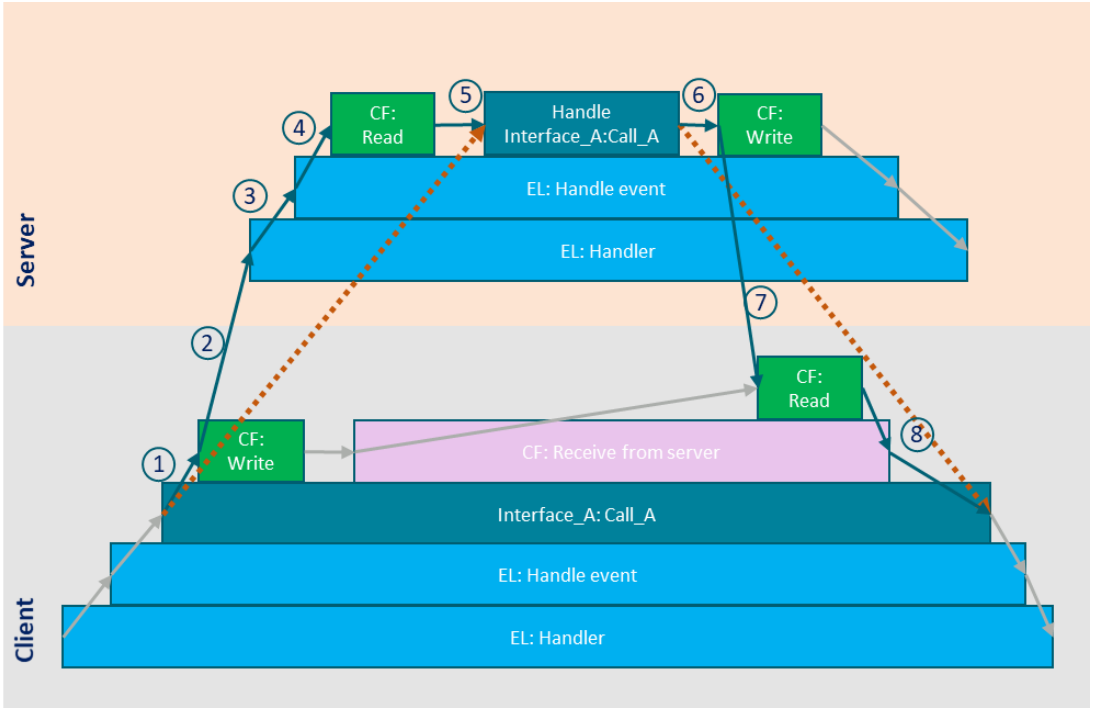

Figure 17. Example of a blocking call through CORBA (IDL, CF, EL).

In order to identify domain dependencies, i.e., a call from the client to the server, it is essential to be able to connect the IDL stubs from the client to the Object adapter and the skeleton interface on the server side. However, in the CORBA case not every IDL call has a unique identifier that can be used to relate events (i.e., infer dependencies). The solution we opted for is to utilize the underlying ORB infrastructure and obtain event IDs from the message passing mechanism. To create the association between the ORB infrastructure and the IDL, we chose to extend the **IDL generators** (stubs and skeleton) with additions that provide information for relating the IDL **calls** with **messages** transmitted over the CF infrastructure and therefore enabling their **identification** at the server side via the EL component.

In addition, not tracing the underlying ORB infrastructure decreases the accuracy of the analysis. As mentioned above, the event loop can handle single event at a time. If the event loop is busy, events arriving from clients will be parked in the queue and processed as soon as the event handler is released, see Figure 18 – 0, 2, 3. Having the timestamps at application level (ORB object at client and server), will not suffice for high-resolution performance analysis.

For all the above reasons, additional probes have been introduced at the ORB level at (see Figure 15):

a. Communication Framework (CF) for capturing the timestamp of the start of a message (call with object) transmission (packing)
b. Event Loop (EL) for capturing the timestamp an event has been detected in the file descriptor
c. Communication Framework (CF) for capturing the timestamp when the message has started to be unpacked

To support automated model inference of TMSC models that contain adequate information for reasoning over performance **diagnostics**, three components have been instrumented with probes, as depicted in Figure 15: **IDL generator**, **CF**, **EL** components.

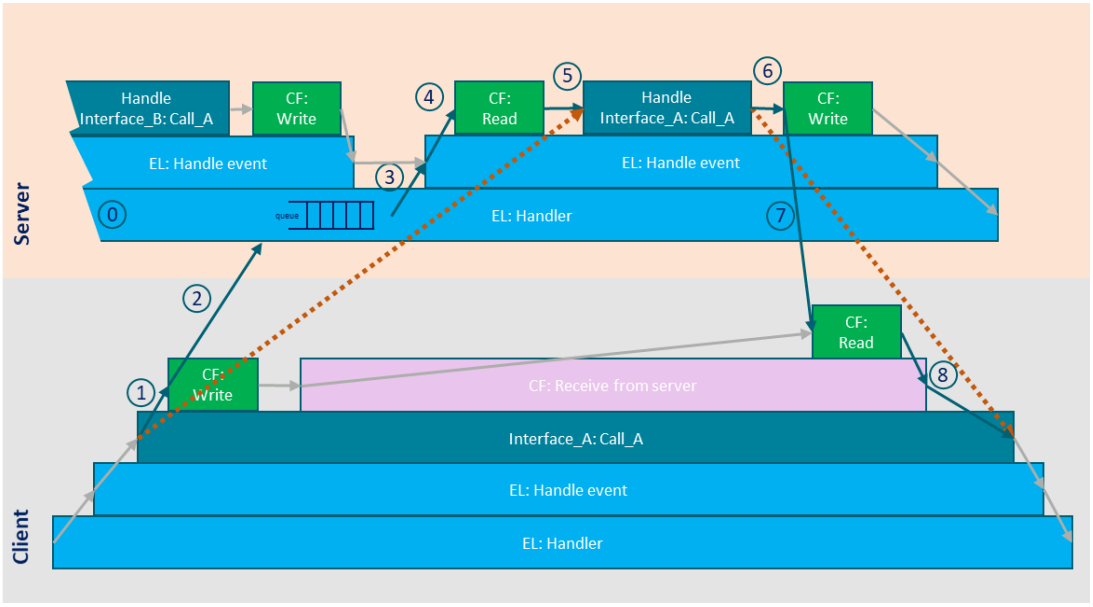

Figure 18. Server side is busy with another request, client-side call is parked in the queue.

LTTng framework: For the system tracing, we opted for the LTTng tracing framework [14]. LTTng is an open-source **toolkit** that can be utilized to trace the Linux kernel, user applications and user libraries at the same time. In the particular case of PPS, the focus is on the user space tracing. We opted for the LTTng tracing framework because it offers a lightweight minimally-intrusive logging mechanism combined with high resolution event logging (ns level), as presented in [15] and [16].

Figure 19 summarizes the events that are traced at all different levels. Depending on the system architecture and its implementation, i.e., what information is exchanged during remote/local function calls, different attributes may be needed to be traced for each probe to infer the execution behavior (TMSC) model. In general, the tracing should be sufficient to enable inference of dependencies between events at any level (i.e., same call stack, same process, same host or different hosts). For inspirational reasons, we provide a list of attributes that were traced per probe setup. This does not imply that the same attributes have to be logged for other systems based on a CORBA architecture.

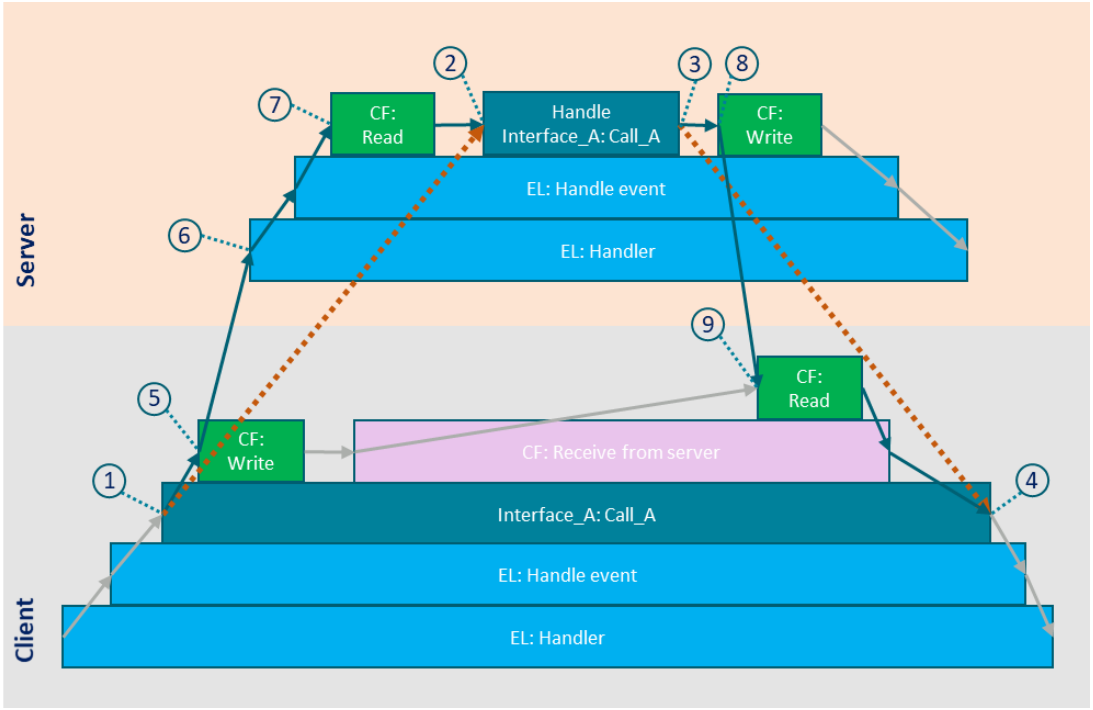

Figure 19. Probes setup at specific event type.

Attributes traced in a specific TWINSCAN use case:
For the IDL calls, the attributes that are logged are interface/method names and CF-server address attributes, which contain server name, process and thread ids. In general, the tracing should be sufficient to enable identification of unique messages and infer the dependencies (connection) between the sender and the receiver. In the TWINSCAN case, the messages do not have a unique ID that facilitates the relation of messages sent (i.e., by the client) to the moment they are handled (i.e., by the server and vice versa). To reconstruct those relations, we need to set up probes on the ORB infrastructure, i.e., CF and EL components. On the communication side, CF has been traced in combination with TCP/IP attributes, i.e., port numbers, socket information to be able to infer the client/server dependencies. Additionally, the required information for the inference of the executors is achieved by tracing the event loop and collecting information at the instantiated stage, such as process and thread IDs. All the traced information is associated with specific timestamps at ns accuracy level. The TMSC model inference of the generated traces will be further discussed in the following section.

Generated traces

After patching the abovementioned components with the probes at specific places, the system can be executed. At this stage, i.e., the Stimulated stage, traces are generated. An example of the trace that is generated for the example in Figure 19, is visualized in Table 1. The parameters traced are the interface and function name of the caller, the type of the event (start (“>”) or end (“<”) event of a function call), a unique timestamp, the executor’s name (i.e., Client, Server) and the message IDs that are used to infer the dependencies between this event and other events. More specifically, the message ID is composed of a prefix, “!” when this event is the source of a message exchange (dependency) or “?” when this event is the target of a message exchange (dependency) and the suffix which is a unique identifier per message exchange and found in both source and target events. For instance, in the first row of the Table 1 the start event of the *EL Handler* function is the source of a message exchange since the message ID has the “!” prefix. The target of this message exchange can be found in row 2, where the start event of the function *EL handle event* is the target of that message

since its prefix is "?" and the suffix matches the suffix of the message exchange in row 1, i.e., *RP_A0.* The event in row 2 is not only acting as target event of a message exchange, but is also the source of another message exchange, i.e., *!RP_A1*. It is apparent that an event can act as source or target of multiple message exchanges.

Table 1. Trace file of the example in Figure 19.

Time Stamp	Executor	Start/End	Interface	Function name	Message ID		
100100	Client	>	EL	Handler	!RP_A0		
100200	Client	>	EL	Handle event	?RP_A0	!RP_A1	
100300	Client	>	Interface_A	Call_A	?RP_A1	!M0	!D_call0
100400	Client	>	CF	Write	?M0	!M1	
100500	Client	<	CF	Write	!RP_A2		
100500	Server	>	EL	Handler	?M1	!ED3	
100600	Client	>	CF	Receive from server	?RP_A2	!RP_A3	
100600	Server	>	EL	Handle event	?ED3	!ED4	
100700	Server	>	CF	Read	?ED4		
100800	Server	<	CF	Read	!ED5		
100900	Server	>	Interface_A	Handle Call_A	?ED5	?D_call0	
101500	Server	<	Interface_A	Handle Call_A	!ED6	!D_call1	
101600	Server	>	CF	Write	?ED6	!ED7	
101700	Client	>	CF	Read	?RP_A3	?ED7	
101700	Server	<	CF	Write	!RP_7		
101800	Client	<	CF	Read	!M2		
101900	Client	<	CF	Receive from server	?M2	!M3	
102000	Server	<	EL	Handle event	?RP_7	!RP_8	
102100	Client	<	Interface_A	Call_A	?M3	!RP_A4	?D_call1
102200	Client	<	EL	Handle event	?RP_A4	!RP_A5	
102200	Server	<	EL	Handler	?RP_8		
102300	Client	<	EL	Handler	?RP_A5		

6.2 TMSC and dependency classification

As already shown in Figure 18, there can be delays on execution of functions due to the services being shared among different applications. The event loop is busy executing other services that may belong to different activities. As activity, we define a set of events and corresponding dependencies that compose a system functionality [17]. Considering the definition of a TMSC as presented in Section 4.2, $TMSC = (E, \rightarrow, m)$, and the definition of the TMSC path as presented in [5], an *activity* TMSC is the set $\{\}$ of paths of the TMSC, defined as follows: $TMSC_a \subseteq TMSC, TMSC_a = \{ e_b \longrightarrow_c^{+i} e_e \}$, where $TMSC_a$ is the activity defined in the TMSC formalism, e_b is the **begin** event of the activity, e_e the **end** event of the activity and $\longrightarrow_c^{+i}$ is the i^{th} path that contains the events and the related dependencies that lead from e_b to e_e, respectively. It is apparent that $e_b \longrightarrow_c^{+} e_e \in TMSC$. The dependencies that belong to an activity are called *activity dependencies* and denote data or control dependencies (implemented order dependency). An activity dependency can be assigned to exactly one activity.

Figure 20 is an extended version of Figure 18, where it is shown that the execution of a function is delayed due to service sharing. Figure 20 visualizes two clients, *Client 1* and *Client 2*. For this example, let's define that *Client 1* belongs to *Activity 1* and *Client 2* to *Activity 2*. The activity dependencies that belong to *Activity 1* are coloured blue and the dependencies that belong to *Activity 2* are coloured orange. While those two activities are data independent and the execution of one should not delay the execution of the other, practically they use services that are deployed on the same lifeline, the Server lifeline. As mentioned in Section 6.1.2 and depicted in Figure 18, if the server side is busy with another request, a client-side call is parked in the queue. On the Server lifeline the two activities are connected via a grey-colored dependency. This dependency here denotes the scheduling aspect of the two handle events and is called a *Schedule dependency*.

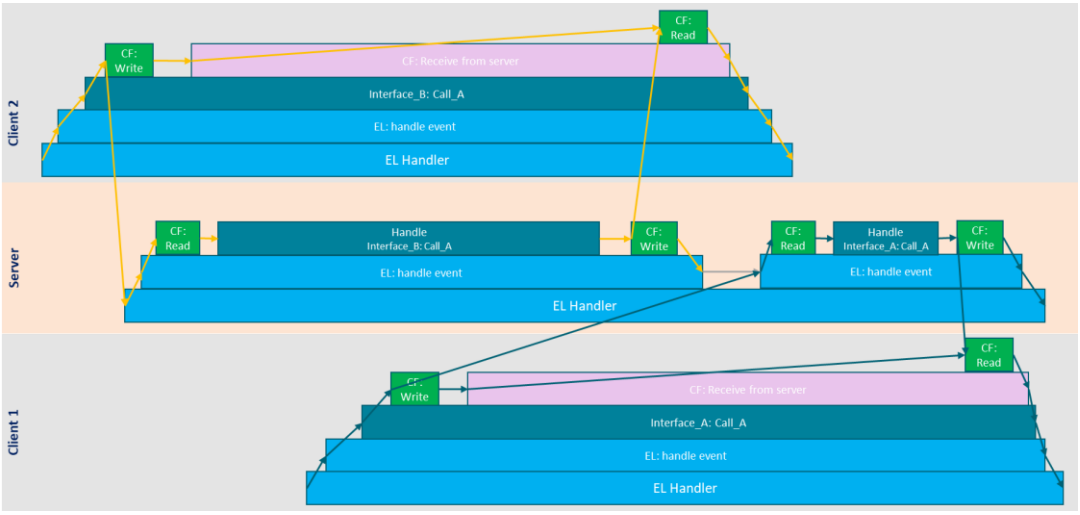

Figure 20. TMSC containing 2 activities which use services deployed on the same Server component.

Summarizing, a TMSC contains several dependencies which are classified as activity and schedule dependencies. Activity dependencies are dependencies between two events that denotes data relation or control, while the schedule dependencies denote schedule relations, i.e., how the event was scheduled by the execution platform, meaning that there is no data dependency/flow.

The visualization of the TMSC and the related activities may support different rendering strategies. Depending on the purpose, the strategy may involve different path definitions. Next, we define for a **begin** and an **end event** three different definition for paths, the *activity,* the *causal* and the *activity causal* path.

Activity path: This path contains only the activity dependencies that belong to a specified activity. This denotes the functionality of the activity by providing the events and function calls, as well as the dependencies and the execution flow of the activity.

Causal path: This path may contain dependencies that belong to multiple activities and/or schedule dependencies. In other words, it does not impose any restriction on the type of dependencies to be part of the path.

Causal activity path: This path is the causal path as defined above, but it contains only the dependencies that are part of the specified activity. The (sub-)paths that belong to the causal path and are **not part** of the activity, are projected out to schedule dependencies, see Figure 23.

Figure 21 shows two activities using the “Server” lifeline to execute its *Interface_B:Call_B* function/service. The **activity path** for both activities is drawn. By visually inspecting Figure 21 we observe that there is no overlap between the colored paths (something that is expected from the definition: activity dependencies can be assigned to a single activity).

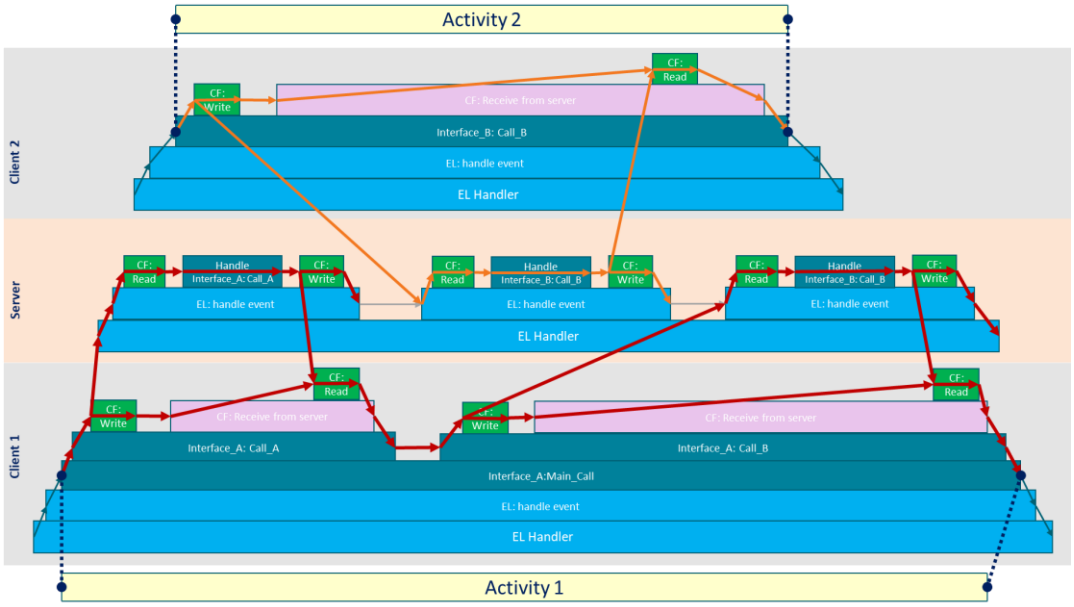

Figure 21. Activity paths for activities 1 and 2. The dotted lines visualize the begin and end events for both activities. The paths are drawn by finding all the paths that lead from the begin to the end event of the activity, excluding dependencies that (a) belong to different activities and (b) schedule dependencies.

Figure 22 shows the **causal path** for Activity 1. This graph can help visualizing that in the causal path of Activity 1, there are dependencies belonging to different activities denoting that in this situation there is service contention.

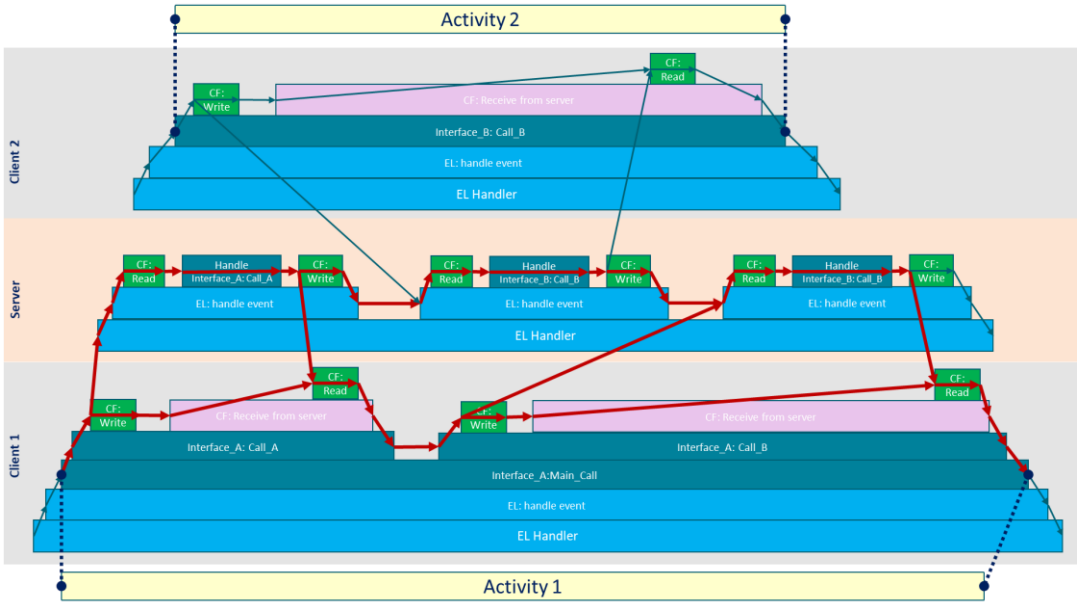

Figure 22. Causal path for Activity 1. The path is drawn by finding all the paths that lead from begin to the end event of the Activity 1, **including** all dependencies.

Figure 23 shows the **causal activity path** for Activity 1. The dependencies that are not part of Activity 1 are projected out. In this case the two schedule dependencies (grey-colored in Figure 21) and the call stack between them are projected out to the single (dotted) schedule dependency.

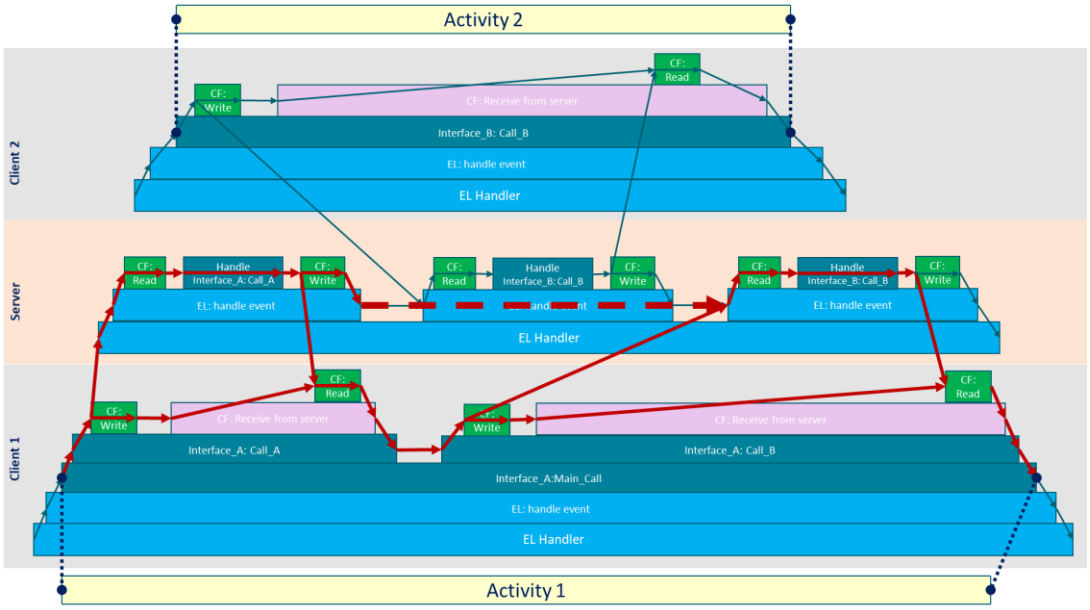

Figure 23. Causal Activity path for Activity 1. The path is drawn by finding all the paths that lead from the begin to the end event of Activity 1, **projecting out** the dependencies/paths that do not belong to the Activity 1 to the dotted schedule dependency.

6.3 Model inference

The refined TMSC metamodel that has been presented in Section 4.3 in combination with the lifecycle software architecture models are used as input together with the generated system traces to infer a TMSC model (Instantiated stage), see Figure 24. The TMSC model is used as input for performance analysis and the root-cause of a performance analysis can be directly bound to a specific artifact/mapping relation of the *Domain specific Lifecycle Software Architecture*, see “refer” arrow in in Figure 24.

The lifecycle software architecture models contain the mapping relations between artifacts. As discussed in Section 5.2, the lifecycle software architecture models of the Specified, Implemented and Instantiated stages contain artifacts that directly relate to model elements in the TMSC meta-model. The encoding of the relation between a component and an executor is captured in the lifecycle software architecture, and therefore the TMSC model inference requires the combination of TMSC and lifecycle software architecture models. The encoding plays an important role in relating observed behavior to artifacts in any of the development lifecycle stages and therefore observations on the stimulated stage can be automatically related to component, interface and implementation artifacts. Considering the meta-models presented in Section 4.3, there is a one-to-one relationship between an executor and a thread. The TMSC swim lanes represent executors as well as the threads as represented in the platform meta-model.

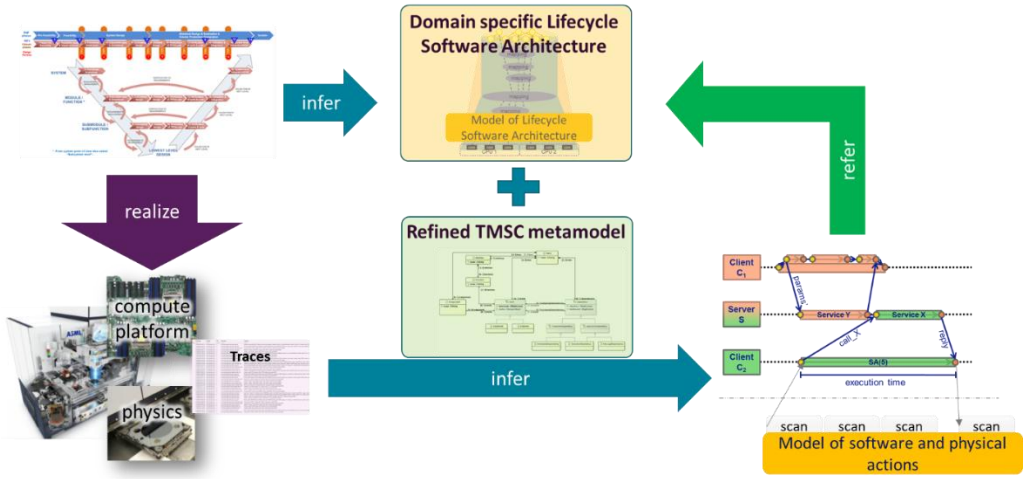

Figure 24. Automated inference of TMSC models.

TWINSCAN use case: The trace in Table 1 and the visualization configuration (i.e., color, arrow styles, etc.) are used as input to the PPS tooling [4]. The PPS tool parses the trace and the configuration and infers the TMSC model and its visualization. To create reasoning over execution platform aspects (i.e., resource usage), additional tracing containing execution platform runtime behavior is required (i.e., the atop tool [18]). Such frameworks capture thread and process IDs along with their mapping on specific processing units (i.e., CPU/core), as well as the utilization and load of the execution platform resources. As mentioned above, the specified trace contains information **on** the process and thread IDs per event probe trace. Using this information and considering the mapping between executor and thread as modelled in the meta-model in Figure 13, we are able to connect TMSC views alongside with platform execution utilization and resource load views which may help in investigation of the root cause of a performance anomaly.

7 Diagnostics and visualization

At this stage the TMSC models are inferred. The TMSC models, as explained in Chapter 4, are generic meaning that from now on we could abstract from specific system implementation and design choices. All analysis and visualization techniques on TMSC models are generic. As soon as the time bounds are calculated (domain specific) they can be used without additional modification per use case.

PPS supports various analysis techniques, such as slack analysis, critical path analysis (CPA), and root cause analysis (RCA). These analysis techniques require that a time bound for each dependency is known, representing the time needed to execute the underlying workload. In the following section, we discuss the time bound computation and present each of the analysis techniques as a straightforward calculation metric based on the time bounds calculations.

7.1 Time bound calculation

Time bound: The time bound of a dependency represents the *minimum* workload of the underlying execution. Given a dependency *d* from event e_A to event e_B , e_B can start at earliest $t(e_A)$ plus the specified time bound (tb): $t(e_B) \geq t(e_A) + tb(d)$. The time bound on the same timeline represents the time that the execution is active (i.e., on CPU time), while on different timelines it represents the communication delay between caller and callee.

In TWINSCAN use case (time bound heuristic): The computation of time bounds for specific dependencies is a challenging and domain (implementation) dependent task. As mentioned above, time bounds can be computed in various ways, for example using statistical analysis of specific types of dependencies or domain knowledge about the execution platform. To illustrate, consider example (a) of Figure 25, where we consider the communication between components via message passing, captured by dependency d_2. This is a typical communication duration dependency and a generic time bound could be assigned based on experiments and data analyses on the duration, while also considering domain knowledge over the communication protocol, i.e., local *vs.* remote calls may have different time bounds, etc.
In the TWINSCAN use case, for the computation of the time bound, we consider three global time bound durations related to the execution platform architecture, since it is not always possible to obtain timestamps for all events occurring in the system.

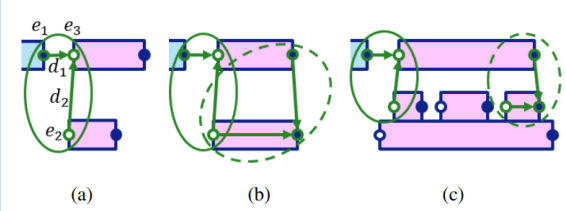

Figure 25. Heuristic for message exchange.

Each of the three examples in Figure 25 shows different types of dependencies for which the time bounds need to be calculated. At this point we utilize domain knowledge to define rules and their exceptions which can be used to calculate the time bound of any dependency that can be observed at the TWINSCAN system. In total four rules have been defined and are described below.
Rule 1 – The time bound for dependencies between events on the same lifeline is equal to the dependency duration as observed in the trace, *tb = d,* where *tb* is the time bound and *d* the dependency

duration. The main assumption considered for this rule is that dependencies between events on the same lifeline are executed in the fastest possible way by the processing unit(s) (i.e., not considering the wait-for-CPU-time due to scheduling in and out). All those execution dependencies are marked with number 1 in Figure 26.

Rule 2 – An exception to rule 1 is the "Receive from server" call, marked as 2 in Figure 26. The *Receive from server* call represents the waiting part (of its parent in the call stack) of a blocking call (i.e., Interface_A: Call_A). The duration of the dependency from the start of the receive until the start of the read is dictated by the moment the reply (message) arrives from the callee, modelling the waiting time in the blocking call. Therefore, the time bound is not equal to the duration of the dependency (since it may contain waiting time). This duration is dependent on the OS/platform and it is determined by experiments. In in this case, this duration was set to a specific value, i.e., 15us. The time bound is computed as the minimum value between this constant number (i.e., 15us) and the dependency duration, $tb = min(15, d)$.

Rule 3 – Another exception to rule 1 is the dependency between two call stacks. In this case, the time bound is equal to the minimum time needed to switch to handling the next event in the event loop queue and depends on the execution platform. Such an example is the dependency marked as 3 in Figure 26.

Rule 4 – The time bound for dependencies between operations executing in different lifelines (i.e., on different executors), marked as 4 in Figure 22, is equal to the minimum value between all communication dependencies durations between the two involved components.

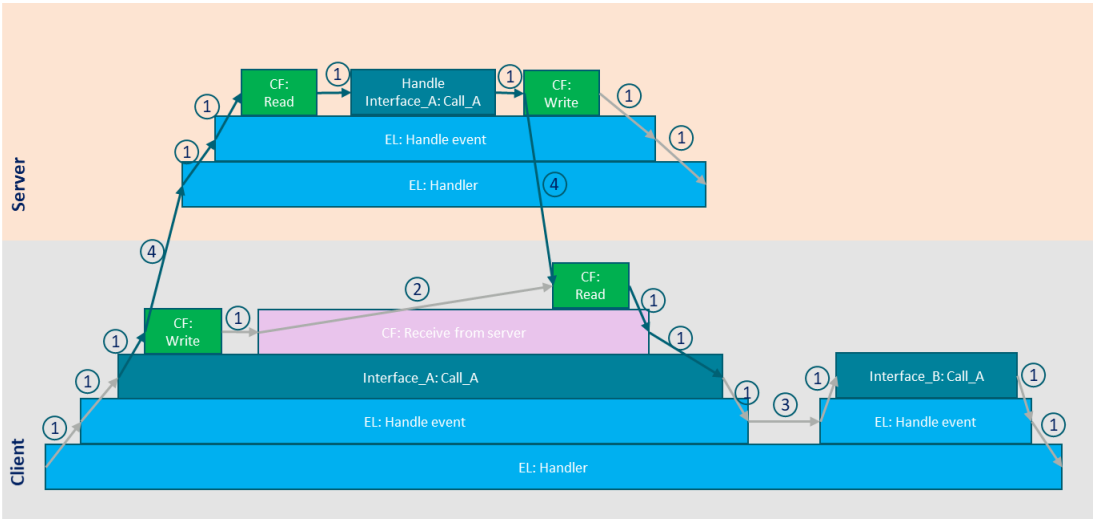

Figure 26. Time bound calculations and dependencies.

7.2 Critical path analysis

The critical path analysis algorithm starts with the classification of dependencies between events to critical and non-critical. As discussed in [5], two events e_A and e_B in a TMSC can be related with a timing bound *tb,* implying that event e_B can be executed at earliest *tb* time units after the event e_A. Any dependency between those two events in a TMSC model is considered as critical when $t(e_A) + tb = t(e_B)$, and as non-critical for the cases that $t(e_A) + tb < t(e_B)$, where function *t(e)* returns the timestamp of an event *e.* Thus, a dependency is critical when its time bound is equal to the dependency duration and non-critical when time bound value is less than the dependency duration.

Let us consider the example of a critical path analysis as visualized in Figure 27. The critical path between two events is the transitive closure [19] of all critical dependencies forming a path that leads from one event to the other.

Starting from the end of the *Interface_A: Call_A* function execution, the closure of incoming dependencies that are critical are added to the path. In this case, there is only one dependency, A, which is critical, based on the definition above, where $tb(A) = d(A)$. Likewise, for dependency B. At the start of the read function execution, there exist two incoming dependencies, C and D. For dependency C it is derived that $tb(C) = d(C)$ and therefore it is critical. For dependency D: $tb(D) = 15us < d(D)$ and therefore this dependency is not critical. The dependencies E, F, G, H, K, L, M are critical for the same reason as dependency A and B and dependency I is critical for the same reason as dependency C.

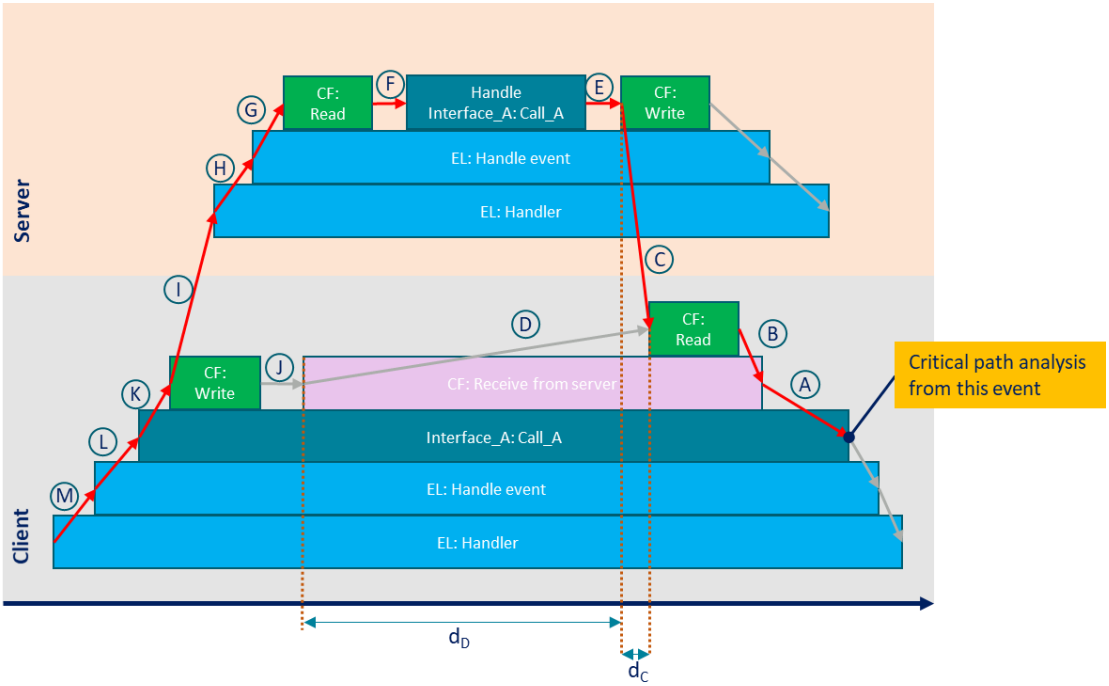

Figure 27. Critical path analysis example.

7.3 Slack analysis

Slack analysis provides the maximum time that each event can be delayed in a TMSC without delaying the starting time of any succeeding event, given the TMSC dependencies. Similar to the critical path analysis we select a reference event (i.e., KPI ending event), visualized in Figure 28. The slack will be computed for all events that precede this reference event.

Consider a $TMSC = (E, \rightarrow, m)$ and an event $e_r \in E$ as reference event, where E is a set of events, $\rightarrow$ defines a timing constraint relation between events, m is a partial function that maps the timing constraint relations to messages. Any preceding event $e \in E$, for which there exists a path between e and e_r ($e \rightarrow^+ e_r$), has a slack $S(e, e_r)$ with respect to the reference event. The slack of each preceding event is defined as the maximum time that this event can be delayed without delaying the reference event e_r, i.e., becoming critical. From the definition of the slack, we have that $S(e_r, e_r) = 0$. At initialization, we consider that for each event e that belongs to the TMSC, except the reference event e_r, the slack is set to infinite, $S(e, e_r) = \infty, e \in (E \setminus e_r)$. Starting from the reference event e_r we consider all incoming dependencies, and we compute the slack for the starting events of those dependencies. The slack of the source event of the dependency D is computed as the value of the dependency duration $d(D)$ minus the time bound $td(D)$ of the dependency plus the slack of the

successor event (dependency destination). Since an event may be the source of multiple dependencies, the slack for such an event is computed as the minimum of all slacks computed for this event. Thus, $S(e, e_r) = \min\{d(D) - tb(D) + S(f, e_r), \forall\, D = e \rightarrow f\}$. As depicted in Figure 24, by applying the abovementioned algorithm and based on the time bound values as computed in previous section, we observe that events that are on the critical path have slack 0, since a possible delay on those will impact the execution time of the reference event. The events that are not part of a path that leads to the reference event e_r, have slack equal to ∞. The start of the *CF:Receive form server* has slack that equals $S = d(J) - tb(J)$. The time bound for this dependency is set to 15us, as discussed above.

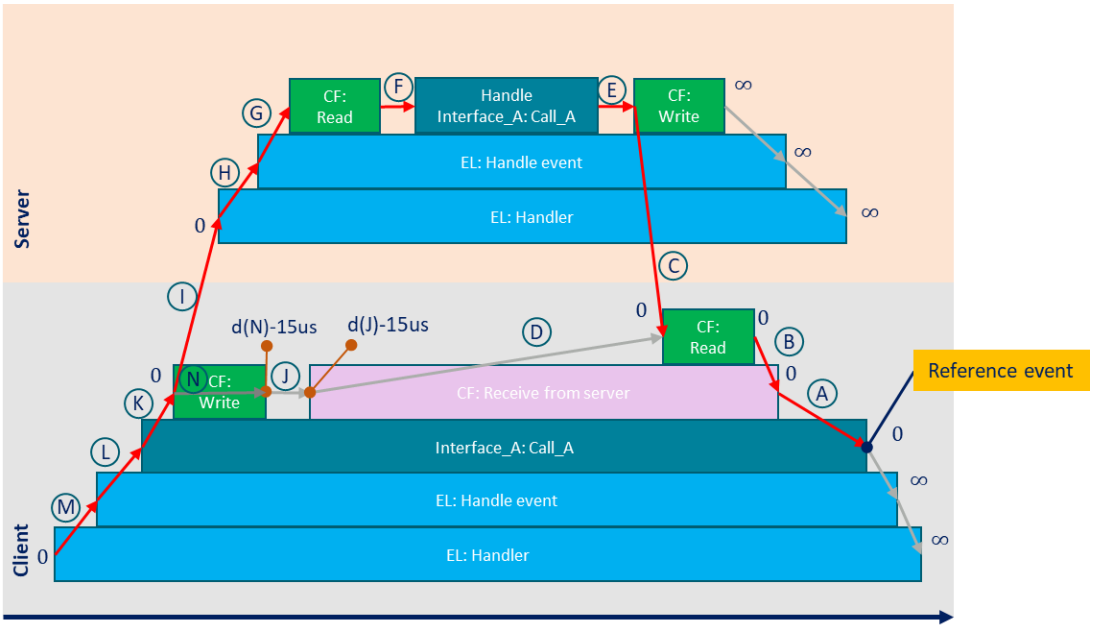

Figure 28. Slack analysis example.

7.4 Model comparison

In the previous section, we provided an overview of the fundamental analysis techniques supported by PPS. However, PPS is capable of performing more complex analyses, which we will delve into in this section. These advanced analysis techniques predominately rely on model comparison. The primary aim of model comparison is to automatically identify structural[1] differences between multiple (possibly large) models – a task that is hard, time-consuming, and tedious when done manually. Take, for instance, the TMSC models shown in Figure 29. While these models seem to be visually very different, they are structurally nearly identical, with about 99.8% of their dependencies being equivalent. Notably, the sole structural differences between these models are small areas highlighted in red, on the top left, in Figure 29. Subsequently, in this section, we initially introduce the model comparison

[1] In a TMCS, the structure pertains to the relations between events, with no consideration given to the duration of dependencies.

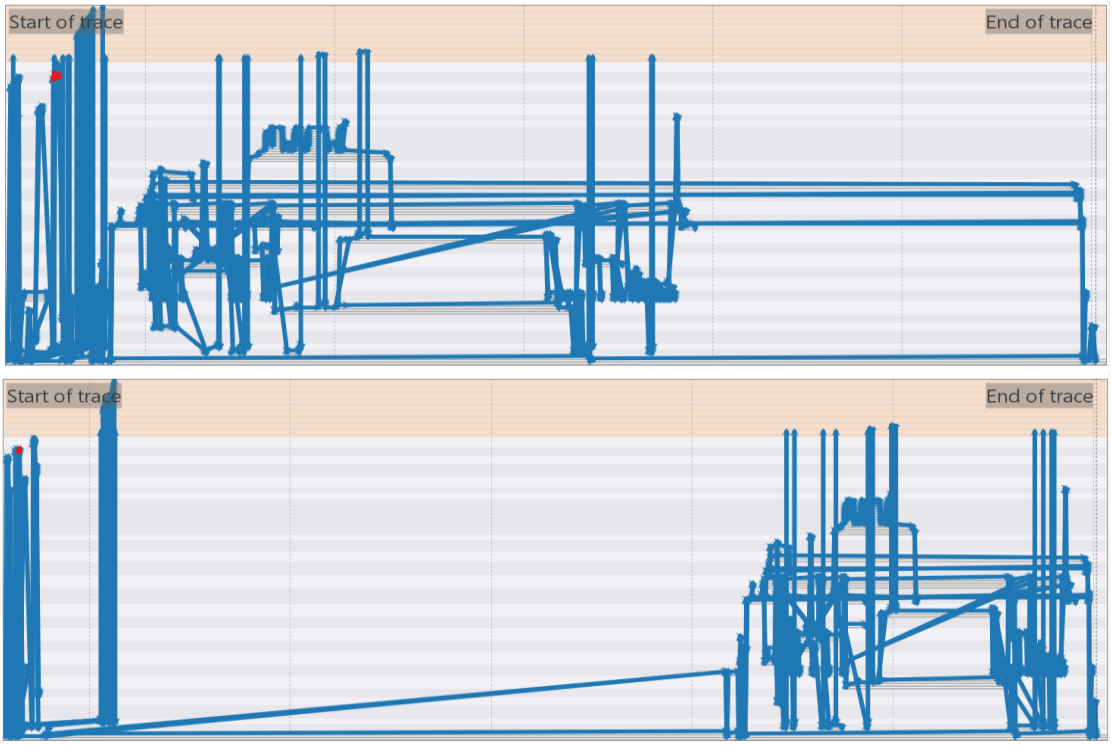

Figure 29. Two compared TMSC models. Although the models look different, they only have minor structural differences, highlighted in the red.

algorithms employed in PPS. Following this, we introduce the potential use cases of employing model comparison.

7.4.1 Model comparison algorithms

PPS features different model comparison algorithms, each having different properties making them suitable for adoption in different scenarios. Given two TMSC models, these algorithms aim at finding the maximum possible overlap between models by matching equivalent events and dependencies, according to the Definitions 1 and 2. In the rest of this section, we introduce the algorithms and further elaborate on their differences.

Definition 1: When excluding time, a TMSC may have several events with the following identical properties in each lifecycle stage (explained in Section 5.2):

- **SPECIFIED:** type, operation (Figure 13), and component (Figure 13)
- **IMPLEMENTED:** type, function (Figure 13), and component
- **INSTANTIATED:** type, function, and lifeline executor (Figure 13)

Please note that, matching events on implemented stage includes/implies matching on specified stage and similarly matching events on instantiated includes/implies matching on implemented stage. This means that when for example two events matches (having identical properties on implemented stage, they also matches on specified stage.

Definition 2: When comparing two TMSC models,

- two events are called *equivalent* if they have identical properties stated in Definition 1;
- two dependencies are called *equivalent* if they have the identical source/target events and type (e.g., execution, request, reply).

a. Equivalence matcher

Two given graphs are called to be *isomorphic equivalent* if they are structurally identical, i.e., having the same set of events/dependencies. To check this, the graphs should have the same size and every event/dependency of one graph should be matched with exactly one event/dependency in the other graph. So, the equivalence matcher traverses the graphs in a topological order[2], assessing if the observed dependencies in both graphs are equivalent (according to Definition 2). Upon encountering inequivalent pair of dependencies, the algorithm terminates the matching process, yielding a false result. Conversely, successful completion results in a true outcome.

b. Isomorphism matcher

In certain use cases, the user may seek to understand the structural variances between the provided models, i.e., essentially conducting a partial comparison. While Equivalence matcher offers a rapid comparison, it is limited to a binary process without providing partial comparison. To address this, the Isomorphism matcher is developed as an extension of the Equivalence matcher. The key difference lies in its continuous exploration even when encountering inequivalent pairs of dependencies between the models. So, unlike the Equivalence matcher, the Isomorphism matcher does not stop. Instead, it tries to match as many dependencies as possible.

Definition 3: An event which has unique properties stated in Definition 1, is called a ***unique*** event.

To begin the matching process, the Isomorphism matcher requires reliable event pairs as starting points. Therefore, it has a preliminary step to identify such event pairs using any of the following ways:

- Find equivalent event pairs just among all *root* and *leaf* events of the models
- Find equivalent event pairs among all *unique* events of the models (according to Definition 3)

Following this step, the algorithm iteratively selects an event pair from the queue. It thoroughly explores both forward and backward paths originating from this selected pair within the graphs. Traversing through these paths in their topological order, the algorithm can match the observed equivalent dependencies until encountering an inequivalent pair of dependencies. Subsequently, it continues the same process with the next starting event pair from the queue until the queue is emptied.

While the algorithm tries to match as many dependencies as possible across various graph paths, it may revisit certain events multiple times. This repetition introduces a risk of matching an event multiple times, once in every visit, leading to uncertainties regarding the accuracy of the matched event pairs. To alleviate any uncertainty, the algorithm stops further progression along that path and eliminates the already matched event from the matched event pairs.

[2] Refers to arranging events of a TMSC in linear order such that event A comes before event B in the ordering if there is a dependency between A and B.

c. Walkinshaw matcher

Another approach is to adopt state-of-the-art (heuristic) algorithms developed and deployed successfully in other domains. For instance, various structural model comparison algorithms have been proposed in the literature for software regression testing, i.e., comparing software behaviors modeled as state machines before and after software upgrades. The state-of-the-art algorithm, developed by Walkinshaw and Bogdanov [20], have multiple variants, providing trade-off between time and quality, i.e., the shorter the matching duration the lower the matching quality. This algorithm is generalized, to support a wide range of state machine model representations using generalized Labeled Transition System (gLTS), and implemented in an open-source framework, called *gLTSdiff* [3] [21]. The formalism of the gLTS model is given in the definition below.

Definition 4 in [21]. A gLTS is a 5-tuple (S, $\mathbb{S}$, P, $\mathbb{T}$, T), where S is a finite set of states, $\mathbb{S}$ is a set of state properties, $P : S \rightarrow \mathbb{S}$ is a function that assigns properties to states, $\mathbb{T}$ is a set of transition properties, and $T \subseteq S \times \mathbb{T} \times S$ is a finite set of transitions.

Given two gLTS models as input, the algorithm in *gLTSdiff* has two main phases to structurally compare the models, which are briefly explained below.

1. Compute similarity scores for all pairs of states of the two gLTS models being compared. This step has two variants, either *local* or *global* scoring. A *local* scoring considers only the overlap in directly connected incoming and outgoing transitions of the states. It is extended to a global scoring by recursively considering all context, using an attenuation factor to ensure that closer context has more impact on the score than more distant context.
2. Use the scores to heuristically compute a matching between states of two gLTSs based on landmarks, a percentage of the highest scoring pairs that score at least some factor better than any other pair, with a fallback to the initial states. The most obviously equivalent state pairs are matched first and these are then used to match the surrounding areas, rejecting any remaining conflicting state pairs. The next-best remaining state pair is then selected and matched until no state pairs are left to consider.

Now, the main step toward adopting the algorithms in *gLTSdiff* in PPS is to transform the TMSC model into a structurally equivalent gLTS model. Following the formal definition of the TMSC meta-model in Section 4.2 and Definition 4, the straightforward transformation of a TMSC model to a gLTS model is to define a new state/transition in the gLTS for each event/dependency of the TMSC. The states/transitions inherit the properties of their corresponding event/dependency. These properties (domain knowledge) are used during scoring phase to prevent the matching of unrelated states, e.g., states associated with different event types (entry and exit), by allocating them $-\infty$ scores to ensure they are excluded from matching in the subsequent phase. An example of such transformation for the TMSC model in Figure 27 is shown in Figure 30.

[3] The gLTSdiff framework is available in https://github.com/TNO/gLTSdiff.

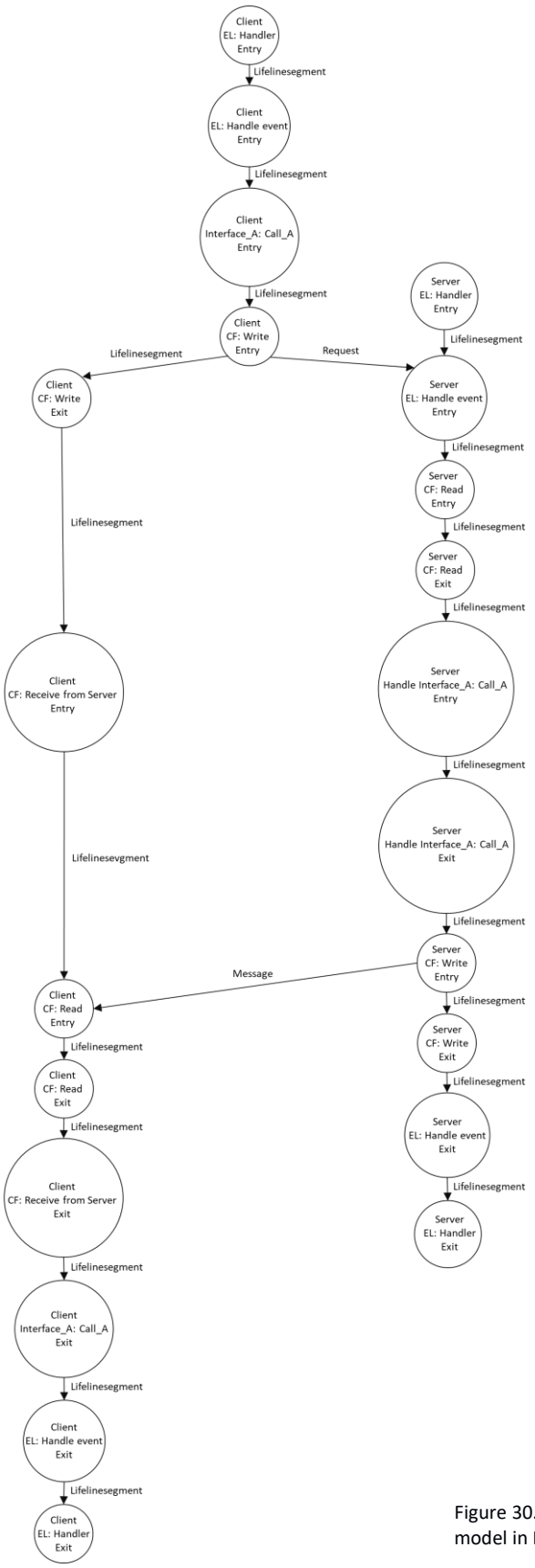

Figure 30. The gLTS model corresponding to the TMSC model in Figure 27.

d. Service matcher

So far, all introduced algorithms try to match events in the context of the whole TMSCs. However, this may negatively affect the matching speed and quality as matching events from numerous possibilities is time consuming and error prone. Instead, the events could be matched in a much smaller and more relevant contexts which can hugely reduce matching possibilities and therefore enhance both matching speed and quality. Following this idea, we introduce a novel algorithm called Service matcher that matches events in the context of call stacks (services). This matcher has two main steps:

1. *Identifying all unique call stacks*: A unique call stack comprises a set of exclusive function calls within each software component. In the context of the ASML use case, we define the uniqueness of a call stack with respect to its root function call, i.e., the lowest-level execution dependency. An execution dependency is unique if both its entry and exit events are unique. Next, by establishing a collection of unique call stacks, it is straightforward to identify related pairs of call stacks between two TMSCs and subsequently match their events, in a smaller context than the whole TMSCs, using any of the previously described algorithms.

2. *Identifying related non-unique call stacks*: Unique call stacks can be linked to multiple (non-)unique call stacks via various dependency types. After identifying and matching unique call stacks, their associated non-unique call stacks can subsequently be identified. The events within corresponding pairs of call stacks can then be matched using any of the previously described algorithms. This iterative process allows for the correlation of (potentially the majority of) call stacks within TMSCs until no further call stacks can be identified.

e. Hybrid matcher

So far, we have introduced four distinct matching algorithms, each possessing unique characteristics. This section briefly overviews these characteristics to guide users when using each algorithm is beneficial.

All algorithms, except the Equivalence matcher, provide partial matching. The Equivalence matcher, however, excels in speed by stopping the comparison process upon detecting the first difference between given models. Among the remaining three algorithms, the Service matcher exploits domain knowledge on the software execution structure, focusing on matching events within isolated call stack contexts, resulting in fast event matching with high confidence level. However, it may not be able to explore the whole graph, leading to low graph coverage. Conversely, the Walkinshaw and Isomorphism matchers are more generic. The Walkinshaw matcher can match large graphs with potentially achieving high graph coverage at the expense of being computationally expensive. The Isomorphism matcher is positioned in between the Service and Walkinshaw matchers, providing higher coverage than the Service matcher while being less computationally expensive than the Walkinshaw matcher. However, both the Walkinshaw and Isomorphism matchers are prone to errors when matching events in extremely large contexts, which potentially results in incorrect event matchings.

In summary, no single algorithm outperforms the others in terms of matching quality, graph coverage and analysis time. There are opportunities for enhancing matching quality and graph coverage while keeping the time short by devising innovative hybrid algorithms. Such an approach combines the strengths of multiple algorithms, thereby enhancing overall performance. A hybrid algorithm might involve a subset of these algorithms stacked sequentially, where each subsequent algorithm utilizes the output of the preceding one.

Following the above discussion, the Equivalence matcher is only recommended for binary comparisons, when no partial matching is required. Otherwise, it is recommended to always initiate the matching process with the Service matcher. If the graph coverage falls below a specific threshold, the matching

process is recommended to continue with the Isomorphism matcher which is faster than the Walkinshaw matcher. Finally, if the coverage remains unsatisfactory, concluding the matching process with the Walkinshaw algorithm may help to further match any unexplored parts in the models.

Table 2 provides insights into the performance of the comparison algorithms and their various combinations, in terms of their coverage and analysis time, through experiments conducted on two cases with TMSCs of different sizes. Entries highlighted in green indicate the optimum algorithms. In both cases, the Service matcher, as expected, stands out for providing the fastest comparison, achieving coverage rate of 93% and 56% in case 1 and case 2, respectively. On the other hand, the combination of Service + Isomorphism + Walkinshaw matchers delivers the highest coverage, reaching 99% and 96% in case 1 and case 2, respectively.

Table 2. The performance of different comparison algorithms and their various combinations, in terms of graph coverage and analysis time, on two cases with different TMSC sizes. In this table, |E| and |→| represent the number of events and dependencies (relations) in each TMSC.

Matching algorithm	Case 1 T1: |E| = 1968, |→| = 2155 T2: |E| = 2338, |→| = 2545		Case 2 T1: |E| = 9874, |→| = 11080 T2: |E| = 9474 , |→| = 10642	
	#Matched Events/Dependencies	Time	#Matched Events/Dependencies	Time
Service matcher	1853/2022	3.75 s	5263/6002	21.9 s
Isomorphism matcher	1926/2109	34.3 s	9168/10243	22.91 m
Walkinshaw matcher	1919/2101	2.92 h	N/A	N/A
Service + Isomorphism matchers	1926/2109	27 s	9168/10243	2.9 m
Service + Walkinshaw matchers	1919/2101	8.7 m	N/A	N/A
Isomorphism + Walkinshaw matchers	1953/2137	16.8 m	9205/10284	29.6 m
Service + Isomorphism + Walkinshaw matchers	1953/2137	7.46 m	9205/10284	12.2 m

Note: All algorithms introduced in this section are *heuristics,* designed to deliver a good result within a short time frame, yet some of them are prone to errors. Users should bear in mind that while these algorithms are powerful enough to match large graphs, the matching results may not always be entirely accurate. However, despite this limitation, the matching result can effectively pinpoint differences, which can be already very useful for users. Nonetheless, some level of user verification appears inevitable to ensure accuracy and completeness.

7.4.2 Use cases

Now that we introduced different model comparison algorithms, we can overview their potential use cases. In general, we classify the use cases into two major groups, namely *horizontal* and *vertical* comparisons. We will further explain these use cases in the rest of this section.

a. Horizontal comparison

The primary objective of horizontal comparison is to compare models extracted from a single system execution trace. Cyber-physical systems' behavior commonly exhibits a high degree of repetition, where sequences of operations recur regularly. This repetitive behavior can be grouped into iterations, each having its own execution model.

The evaluation of system productivity relies on various key performance metrics. A metric is defined as a system activity with specific budget, i.e., $m = \{TMSC_a, B\}$ where metric m is defined as the collection of the causal-activity TMSC of activity $TMSC_a$ (as defined in Section 6.2) and the budget B measured in a time unit. Budget is the maximum allowed duration between the begin event e_b and end event e_e of $TMSC_a$. In the context of ASML use case, due to the repetitive nature of system behavior, each specified metric may have multiple instances, each representing a specific occurrence within the system execution trace and consequently having its own distinctive execution model. In addition, as both the execution platforms and running tasks are not real time, the system behavior in terms of timing performance is non-deterministic, thereby leading to timing variations between instances of a metric. For example, each wafer exposure encompasses a series of multiple concatenated exposures of different parts of the wafer. All exposures within the same wafer share the same configuration (i.e., recipe) and therefore their execution models are expected to be identical. Nevertheless, not all iterations are entirely identical and some variations in their timing and/or structure may exist. These variations can be explored using model comparison, especially during root-case analysis (RCA) when budget violation occurs for a certain metric instance.

Once an execution trace is successfully imported into PPS, reports for the metrics selected by the user are automatically generated. Through the reports, the user obtains a high-level overview of the durations of all metric instances in a bar chart format. Among all instances, potential outliers with budget violations may exist. A metric instance may experience delays caused by service and/or resource contention. Service contention occurs when an application is unable to execute at its optimal speed, as it is delayed by another application which uses the same services. Resource contention, on the other hand, arises from a shortage of physical resources[4]. For each outlier, RCA can automatically pinpoint dependencies whose duration is significantly longer than normal, ultimately causing budget overruns. To do this, RCA takes three main steps, briefly explained in the following:

1. *Normal time-bound retrieval per dependency*: To effectively apply RCA, knowing the typical time bound of each dependency is crucial. Yet, this information is not available to designers during the design phase; it depends on deployment stage and varies during the system execution. Nonetheless, each dependency can have a sample-set of comparable time-bounds by gathering time-bounds of its counterpart from isomorphic TMSC instances that meet the budget requirements. This assumes that structurally equivalent TMSCs behave functionally the same, therefore the duration of corresponding dependencies can be compared. This

[4] Resource contention can be detected with the help of kernel-space tracings, like CPU utilization.

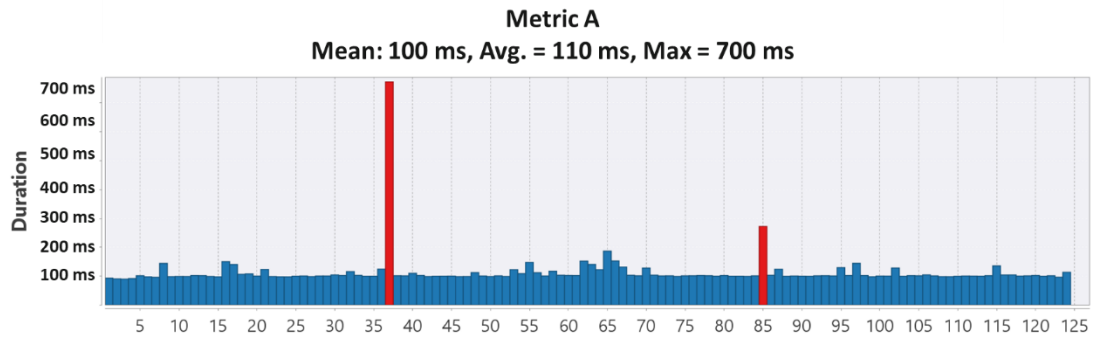

Figure 31. Metric A with 124 instances. The two instances highlighted in red exceed their budget (i.e. outliers).

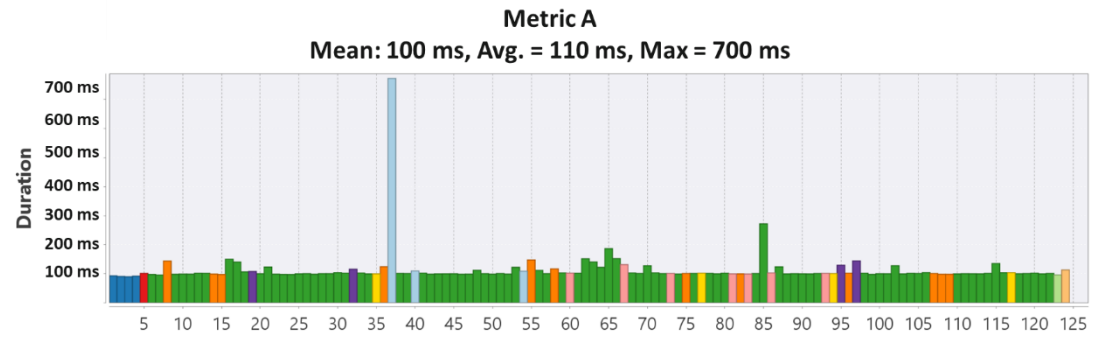

Figure 32. Classification of TMSC model of instances of Metric A in Figure 31 into 10 isomorphic classes. Metric instances of each class are highlighted with a unique color.

implies that the higher the number of isomorphic TMSCs, the richer (in terms of sample size) the sample-set of each dependency.
At present, PPS focuses solely on gathering time-bound samples from isomorphic causal-activity TMSCs. Following the explanation in Section 6.2, such a TMSC contains subsets of schedule and activity dependencies of the original model that are part of the causal paths. Any causal path can potentially be critical (as explained in Section 7.2), indicating the minimum duration of a metric instance. Therefore, it is imperative to gather sufficient time-bound samples for all dependencies within all causal paths.

2. *Statistical outlier analysis per dependency*: The time-bound sample set enables the application of statistical outlier analysis on each individual dependency. Using this set, a time-bound threshold can be statistically calculated for each dependency using the standard formula of $Q_3 + (Q_3 - Q_1) \times IQR_{factor}$[5]. Consequently, dependencies that have longer time bound than their threshold are considered as critical dependencies.

3. *Root-cause identification:* The objective of root cause algorithm is to identify the smallest set of critical dependencies, such that if their time bound would have been normal, as in the non-violating cases, the budget requirement would not have been violated. The algorithm consists of two primary steps. In the first step, it evaluates the application's makespan by resolving

[5] Q_1 and Q_3 are the first and third quartiles of the sample set and $Q_3 - Q_1$ is the interquartile range (*IQR*) including half of the samples. Samples smaller than $Q_1 - IQR \times IQR_{factor}$ and bigger than $Q_3 + IQR \times IQR_{factor}$ are statically considered as outliers. We set $IQR_{factor} = 3.0$ in the context of ASML use case.

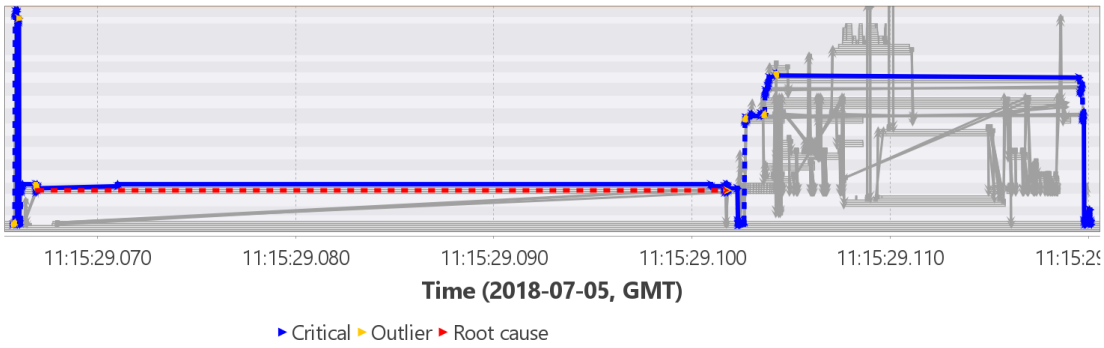

Figure 33. The outcome of applying automated RCA to instance 85.

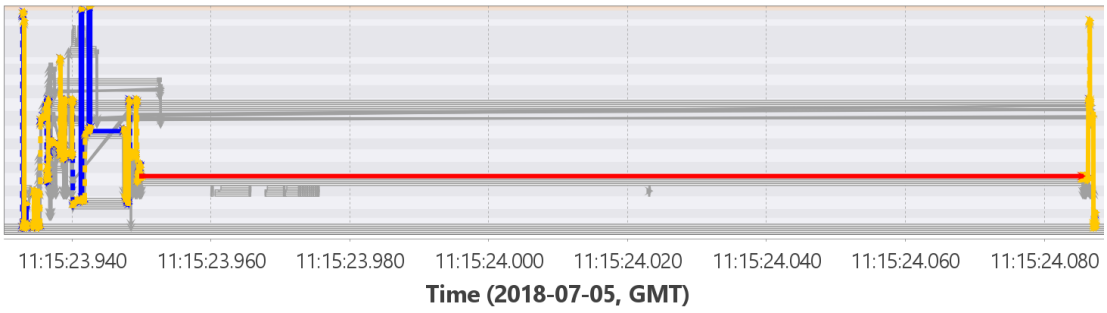

Figure 34. The outcome of applying automated RCA to instance 37.

each critical dependency through substituting its time bound with its computed threshold. These dependencies are then sorted based on their impact in reducing the makespan. In the second step, the algorithm focuses on the critical dependencies within the critical path and selects the one with maximum reduction in the makespan. It checks whether resolving this dependency brings the makespan within the budget constraint. If not, the algorithm iterates in the second step, by selecting the subsequent critical dependencies in a similar manner. Note that the model may have several causal paths from the begin to the end events of the model. So, resolving certain dependency in each iteration may alter the application's critical path which is then considered in the next iteration. The iterative process continues until either the makespan falls within the budget or a feasible solution cannot be found.

For instance, Figure 31 represents the report of a wafer exposure related metric (Metric A), in the ASML use case, containing 124 instances all belonging to a single wafer exposure. Among all metric instances, two instances (37 and 85) highlighted in red exceed the budget constraint. Using the Equivalence matcher, explained in Section 7.4.1.a, TMSC model of all metric instances can be quickly compared structurally and grouped into 10 isomorphic classes. Figure 32 shows the classes with different colors, i.e., each class's members highlighted in the same color.

Instance 85 belongs to the class with the largest number of instances, i.e., 87 instances highlighted in green in Figure 32. Consequently, all dependencies within this instance's TMSC model have 87 time-bound samples, providing sufficient data for accurate statistical outlier analysis. The outcome of applying RCA to this instance is presented in Figure 33. Within this visualization, a few dependencies are highlighted in yellow, indicating outlier dependencies. Among them, a single root cause is identified and highlighted in red.

The other outlier, instance 37 in Figure 31, is part of the class with only two siblings, highlighted in light blue in Figure 32. Consequently, all dependencies within this instance's TMSC model have only 3 time-

bound samples, rendering the data insufficient for accurate statistical outlier analysis. The outcome of applying RCA to this instance is presented in Figure 34. Within this visualization, numerous dependencies are highlighted in yellow, indicating false positive results in the identification of outlier dependencies. Among them, a single root cause is identified and highlighted in red.

To address these false positive outcomes, it is needed to increase the number of time-bound samples of dependencies. Partial comparison of TMSC instances using alternative comparison algorithms, listed in Section 7.4.1, can facilitate this process. This is based on the assumption that all metric instances belonging to the same wafer/lot share the same configuration/recipe. Therefore, minor structural differences between instances, like TMSC models shown in Figure 29 should not significantly alter the software behavior and therefore timings between non-isomorphic instances are expected to be still comparable.

b. Vertical comparison

The primary objective of the vertical comparison is to compare models extracted from a multiple system captures. A highly relevant use case for this scenario is qualifying software releases during which system-level productivities before and after software release (referred to as Pre and Post) are compared. In the case of productivity loss, model comparison enables users to rapidly identify changes in deployed software artifacts in Post resulting from the software upgrades, such as new function calls and dependencies. Additionally, RCA can be applied to Post in the same manner as in horizontal comparison, but using the time-bound samples collected from Pre, in order to identify dependencies with exceptional duration, most likely as a result of the software upgrades. This information can be crucial for identifying potential productivity issues, maintaining system performance, and planning for future development.

8 Closure

In this report we conclude research work on performance analysis of CPS. We have presented the PPS framework and its application on a state-of-the-art complex CPS, the ASML TWINSCAN system. Due to the confidentiality of the case, the actual architecture implementation could not be shared, but instead the application of the PPS on a conceptually similar architecture, the CORBA, is presented in detail. The aim is to help the reader to understand the aspects that need to be taken into account and addressed for the application of the PPS in a different CPS with different architecture or even when applied on different domain. The critical path, slack and root-cause analysis as well as trace comparison analysis are demonstrated during the application of the PPS method on the performance analysis of the ASML TWINSCAN system. The PPS framework is currently being industrialized by ASML in the form of T-iPPS (TWINSCAN integrated Platform Performance Suite) tool which is an extension of the open-source PPS tool [4]. T-iPPS has been already successfully used by ASML as discussed in [8] and [9] proving the value of the methodology and the supporting tooling that were developed during the Maestro project.

9 References

[1] M. W. Maier, “Architecting principles for systems-of-systems,” *The journal if international on Systems Engineering,* 1998.

[2] P. D. Borches and G. M. Bonnema, “System Evolution Barriers and How to Overcome Them!,” in *8th Conference on Systems Engineering Research (CSER)*, Hoboken, NJ, 2010.

[3] B. van der Sanden, Y. Li, J. van den Aker, B. Akesson, T. Bijlsma, M. Hendriks, K. Triantafyllidis, J. Verriet, J. Voeten and T. Basten, “Model-Driven System-Performance Engineering for Cyber-Physical Systems : Industry Session Paper,” in *2021 International Conference on Embedded Software*, 2021.

[4] “PPS tool,” [Online]. Available: https://ci.tno.nl/gitlab/PPS/PPS_tool.

[5] R. Jonk, “Inferring Timed Message Sequence Charts from Execution Traces of Large-scale Componentbased Software Systems,” TUE, Eindhoven, 2019.

[6] H. Koziolek, “Performance Evaluation of Component-based Software Systems: A Survey,” Elsevier, 2010.

[7] D. Bell, “UML basics: The sequence diagram,” developerWorks, 2004.

[8] N. Roos, “Clearing the critical software path,” *Bits&Chips,* 3 May 2021.

[9] N. Roos, “Taking performance analysis to the system level,” *Bits&Chips,* 22 August 2023.

[10] OMG, “omg.org,” OMG, 02 2021. [Online]. Available: https://www.omg.org/spec/CORBA/3.4/. [Accessed 05 01 2022].

[11] OMG, “Corba Interoperability,” 14 11 2012. [Online]. Available: https://www.omg.org/spec/CORBA/3.4/Interoperability/PDF. [Accessed 05 01 2022].

[12] OMG, “CORBA Interfaces,” 02 02 2021. [Online]. Available: https://www.omg.org/spec/CORBA/3.4/Interfaces/PDF. [Accessed 05 01 2022].

[13] D. C. Schmidt, “Object Interconnections: Programming Asynchronous Method Invocations with CORBA Messaging,” SIGS C++ Report magazine, 1999.

[14] “LTTng,” [Online]. Available: https://lttng.org/. [Accessed 11 02 2021].

[15] M. Desnoyers, “Tracing for Hardware, Driver and Binary Reverse Engineering in Linux”.

[16] M. Desnoyers, “The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux.,” 2006.

[17] B. van der Sanden, J. Bastos, J. Voeten, M. Geilen, M. Reniers and T. Basten, “Compositional specification of functionality and timing of manufacturing systems,” in *Forum on Specification and Design Languages (FDL)*, Bremen, Germany, 2016.

[18] “Atop tool,” [Online]. Available: https://www.atoptool.nl/index.php.

[19] “Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Transitive_closure. [Accessed 29 04 2022].

[20] N. Walkinshaw and K. Bogdanov, “Automated comparison of state-based software models in terms of their language and structure,” *ACM Transactions on Software Engineering and Methodology (TOSEM),* 2013.

[21] D. Hendriks and W. Oortwijn, “gLTSdiff: A Generalized Framework for Structural Comparison of Software Behavior,” MODELS, 2023.

ICT, Strategy & Policy

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

TNO innovation for life