Science of Computer Programming 233 (2024) 103067

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Model based component development and analysis with ComMA

Ivan Kurtev [a,b,*], Jozef Hooman [c], Mathijs Schuts [d,e], Daan van der Munnik [e]

[a] Eindhoven University of Technology, Eindhoven, the Netherlands
[b] Capgemini Engineering, Eindhoven, the Netherlands
[c] TNO-ESI, Eindhoven, the Netherlands
[d] Radboud University, Nijmegen, the Netherlands
[e] Philips, Best, the Netherlands

ARTICLE INFO

Keywords:
Model-driven engineering
Interface modeling
Component modeling
Domain-specific languages
Component-based development

ABSTRACT

The lack of explicit and precise specifications of software interfaces between components often leads to integration issues during development and maintenance. To address this, we have developed a framework named ComMA (Component Modeling and Analysis) that supports model-based engineering of high-tech systems by precisely defining components and their interfaces. The framework is a family of Domain Specific Languages (DSLs) for modeling component interfaces, protocol state machines, time and data constraints, and constraints on relations between events of multiple interfaces. From these models a number of artifacts can be generated automatically to support analysis and various engineering tasks. ComMA has been developed in close collaboration with the Philips IGT business unit that develops minimally-invasive X-ray systems. This paper presents the experience we gained in creating the ComMA framework and its application in industrial practice. We describe and reflect on the technical, organizational and process-related aspects of deploying a non-trivial MDE solution in an industrial setting.

1. Introduction

Contemporary high-tech systems are complex artifacts integrating numerous interacting software and hardware components that may be developed by geographically distributed departments or supplied by different vendors. The lack of explicit and precise specifications of component interfaces potentially leads to problems during component integration. Furthermore, updates of components in systems that are already deployed and operational may also lead to issues due to, for example, unexpected changes in the interface protocol and time behavior. Precise specifications of components and their interfaces are enablers for successful system integration. During system development, proper interface definitions allow checking the compatibility of integrated components. In the later phases of the system life cycle, they may serve as specifications of properties to be continuously monitored to detect inconsistencies due to component changes.

Typically an interface lists its methods (commands) and notifications (events) that form a signature and is sometimes accompanied by an indication of allowed interactions (interface protocol). Time expectations (e.g., reactions to commands and periodicity of events) and constraints on data parameters often stay implicit. Companies may use standard technologies for transparent deployment of interfaces (e.g., ZeroMQ, gRPC) or company-specific solutions, but the interface definitions are often only specified in a document

* Corresponding author.
E-mail address: i.kurtev@tue.nl (I. Kurtev).

https://doi.org/10.1016/j.scico.2023.103067
Received 18 March 2023; Received in revised form 15 November 2023; Accepted 16 November 2023

Available online 22 November 2023

```
signature IVacuum
commands
  void VacuumOn
  void VacuumOff
notifications
  VacuumOK
```

Listing 1: Signature of the IVacuum interface.

in natural text and informal diagrams. This informal nature of interface specification makes it difficult to check that implementations conform to their specification.

In order to address the aforementioned issues, the ComMA (Component Modeling and Analysis) method and tool have been developed to support model-based engineering by precisely defining components and their interfaces.

ComMA has been developed in close collaboration with industry. The initial motivation for creating the ComMA framework originates from the Philips IGT business unit that develops minimally-invasive X-ray systems and is related to initiatives for optimizing the integration process and reducing its costs. A broad interest in the topic was also observed in a number of knowledge exchange meetings attended by several companies from the Dutch high-tech sector where the need for component and interface modeling is clearly recognized.

ComMA is developed following the model-driven engineering approach (MDE) [1]. In a nutshell, the framework consists of a family of domain-specific modeling languages (DSLs) and tools for supporting validation and verification tasks. The interface specification language of ComMA allows engineers to define the interface behavior in a state machine-based notation. Timing constraints are defined as relations between communication events decorated with the admissible time intervals. Data constraints specify conditions over parameter values in one or more events. The component modeling language allows the definition of simple and compound components (containing multiple sub-components as parts) that use several interfaces together. The tool support includes automatic checking of interface models (e.g., detecting deadlocks and race conditions) and generating simulators and tests. In particular, runtime verification is supported by monitoring and checking component execution traces against specifications. Model transformations serve as bridges to other analysis and visualization tools.

The development of ComMA spans more than 6 years (with first steps initiated in 2015–2016). The tooling has been made open source in the CommaSuite® project of the Eclipse Foundation.[1]

This paper presents the experience we gained in developing ComMA and its application in industry. First, a global overview of ComMA is provided in Section 2 including the main languages, the generators and the tooling. Section 3 contains a reflection on the use of MDE. Section 4 discusses related work in the area of interface and component modeling. Concluding remarks, current and future work can be found in Section 5.

2. Overview of the ComMA framework

This section provides an overview of the ComMA framework. The modeling language is presented in Section 2.1. The tasks that generate artifacts from models are discussed in Section 2.2. Section 2.3 describes the available tool support.

2.1. Modeling language

The ComMA project is executed following an industry-as-laboratory approach [2]. This means that tools and techniques are developed in close interaction, typically on a weekly basis, with real industrial projects. One of the main goals is allowing easy application by industrial users. The modeling languages use familiar engineering notations such as state-based specification of the behavioral aspects, commonly found patterns for timing properties, and software architecture description concepts for component models. The languages have been developed iteratively respecting the requests and the feedback from the industrial users.

More concretely, ComMA provides the following domain-specific languages: an interface signature language, described in Section 2.1.1, an interface language, presented in Section 2.1.2, and a component language, introduced in Section 2.1.3. Not shown here is a type language which allows the definition of types that can be used in the models. Supported constructs are enumerations, records, collections and maps. We also omit the details of a few other languages with a supporting role: a language for input and output parameters used in simulations, a language for component execution traces, and a language for specifying generator tasks.

2.1.1. Interface signature

Based on the needs of our industrial partners, ComMA supports client-server architectures where clients can call methods of the server. *Commands* are synchronous methods that need a reply from the server. *Signals* are asynchronous methods without a reply. The server cannot call methods of the client, but it can send asynchronous *notifications*.

The messages that can be exchanged between a client and server will be referred to as *interface events* or *messages*. Events may carry parameters. The events that belong to a given interface form a *signature*. The interface signature language is illustrated in Listing 1 by a simple example of an interface called *IVacuum*, for managing vacuum in a system.

[1] https://eclipse.dev/comma.

```
interface IVacuum version "1.0"

machine VacuumMachine {
    initial state NoVacuum {
        transition trigger: VacuumOn
            do: reply
            next state: Evacuating
    }
    state Evacuating {
        transition
            do: VacuumOK
            next state: Vacuum
    }
    state Vacuum {
        transition trigger: VacuumOff
            do: reply
            next state: NoVacuum
    }
}

timing constraints
VacuumOnResponse
command VacuumOn-[2.0 ms..50.1 ms]-> notification VacuumOK
VacuumOffDelay
command VacuumOff-[..10.4 ms]-> reply to command VacuumOff
```

Listing 2: State machine and timing constraints of the IVacuum interface.

```
component Control
    provided port IControl iControlPort
    required port ITemperature iTemperaturePort
    required port IVacuum iVacuumPort
    required port ISource iSourcePort
```

Listing 3: Interfaces of component Control.

2.1.2. Interface modeling language

The allowed order of exchanged messages between a client and server is defined in the interface model by means of a protocol state machine. In addition, time and data constraints may be defined. Listing 2 contains the state machine of the *IVacuum* interface and a few timing constraints.

The state machine describes a client-server interface from the viewpoint of a server, that is, transitions are triggered by client calls. The *do* part of a transition contains a sequence of actions of the server, which may include assignments to variables, a reply to a command, if-then-else statements, and notifications. Transitions may have guards (not shown in the example). Non-determinism is allowed, e.g., after a client call there may be multiple possible transitions by the server, possibly leading to different responses and states.

The example shows two timing constraints. The first one, *VacuumOnResponse*, states that if command *VacuumOn* is received, the notification *VacuumOK* is expected between 2 and 50.1 milliseconds after the command. Other kinds of constraints are supported, for example, expressing periodic behavior. More details can be found in [3].

2.1.3. Component modeling language

Typically, a software component provides a number of interfaces to its clients and it requires a number of interfaces to use services of other components. As an example, we consider a simple *Control* component that provides interface *IControl* to its clients and uses services from other components via three interfaces, as shown in Listing 3.

In component models, ports are connection points used in the communication between component instances and are always associated to an interface. We distinguish between provided and required ports. A provided port is used by the clients of the component for connecting to and interacting with it according to the port's interface. Required ports are used by the component to call services of other components.

The main purpose of a component model is to define constraints on the order of events sent over the component ports. The construct used to specify this order is called a *functional constraint*. A functional constraint captures an aspect of the complete behavior of the component and is usually restricted to a small subset of the observable events. Component models do not define the complete component behavior in terms of reactions to all possible events in different states. A component implementation is expected to satisfy all the functional constraints defined in a component model. In addition, a component model may define time and data constraints.

Functional constraints have two forms: state-based specifications (known as state-based functional constraints) and expressions that have to evaluate to true for every observed event in the component context (called predicate functional constraints).

Our example *Control* component model handles requests for acquiring an image of a specimen (via *iControlPort*) and controls the vacuum and temperature in the system (via *iVacuumPort* and *iTemperaturePort*). A requirement for the control logic is that image

```
use events
command iControlPort::AcquireImage
iControlPort::reply to command AcquireImage
command iSourcePort::StartAcquisition
iSourcePort::reply to command StartAcquisition

initial state Ready {
    // if vacuum and temperature OK, start acquisition
    transition trigger: iControlPort::AcquireImage
    guard: iVacuumPort in Vacuum and
           iTemperaturePort in TemperatureSet
       do: iControlPort::reply
           iSourcePort::StartAcquisition
       next state: AwaitReply
}
state AwaitReply {
    transition trigger: iSourcePort::reply to command StartAcquisition
        next state: Ready
}
```

Listing 4: Example of a functional constraint of component Control.

```
component ImageAcquisition
    provided port IControl iControlPort1
    required port IProcessing iProcessingPort
parts
    Control control
    Temperature temperature
    Vacuum vacuum
    Acquisition acquisition
connections
    iControlPort1 <-> control::iControlPort
    control::iTemperaturePort <-> temperature::iTemperaturePort
    control::iVacuumPort <-> vacuum::iVacuumPort
    control::iSourcePort <-> acquisition::iSourcePort
    acquisition::iProcessingPort <-> iProcessingPort
```

Listing 5: A compound component with four parts.

acquisition is only possible if vacuum is present, and a certain temperature is reached. In terms of allowed message sequences, the command *AcquireImage* must be observed after the notifications about the correct state of vacuum and temperature, and only then the acquisition can be started. A functional constraint that captures this requirement is shown in Listing 4.

The constraint uses four events listed in the section *use events*. The allowed order of the specified events is given in a state machine similarly to the interface protocol state machines. Events that do not belong to the set defined in *use events* are not restricted. In the example, the transition in state *Ready* is triggered when command *AcquireImage* is observed at port *iControlPort* and the transition guard evaluates to true. Expression *iVacuumPort in Vacuum* is true if the sequence of messages at *iVacuumPort* until the observation of *AcquireImage* has led to state *Vacuum*. Here *Vacuum* is a state in the *IVacuum* interface as shown previously. The same is valid for the other conjunct.

In summary, *AcquireImage* is allowed to occur only if the vacuum and temperature ports are in the right state. This access to interface state information of ports is very convenient. Without it, the functional constraint needs to replicate the sequence of the messages on the two ports that lead to the indicated interface states, which is information already present in the interface specifications.

Component models may also define the internal component structure: its sub-components (parts) and their interactions. In the example of Listing 5, the *Control* component is used in a larger context, namely, component *ImageAcquisition* with four parts. Each part has a name and a type (a component model). The parts have all the provided and required ports defined in their component model. The allowed interactions among the parts are indicated via connections between their ports. For example, the required port *iTemperaturePort* of part *control* is connected to the provided port with the same name belonging to part *temperature*. It is also possible to connect the ports of the containing component to the ports of its parts (observe the connection between port *iControlPort1* and *control::iControlPort*). Similar to simple components, the allowed order of events over the component ports and the ports of its parts is given in functional constraints.

2.2. Generators

The purpose of the ComMA models expressed in the described languages is to specify the intended behavior of the component and interface implementations. The models are used in engineering tasks like producing documentation, validation, and verification. Concerning the verification, an important part of ComMA is the ability to perform runtime monitoring of components to check if

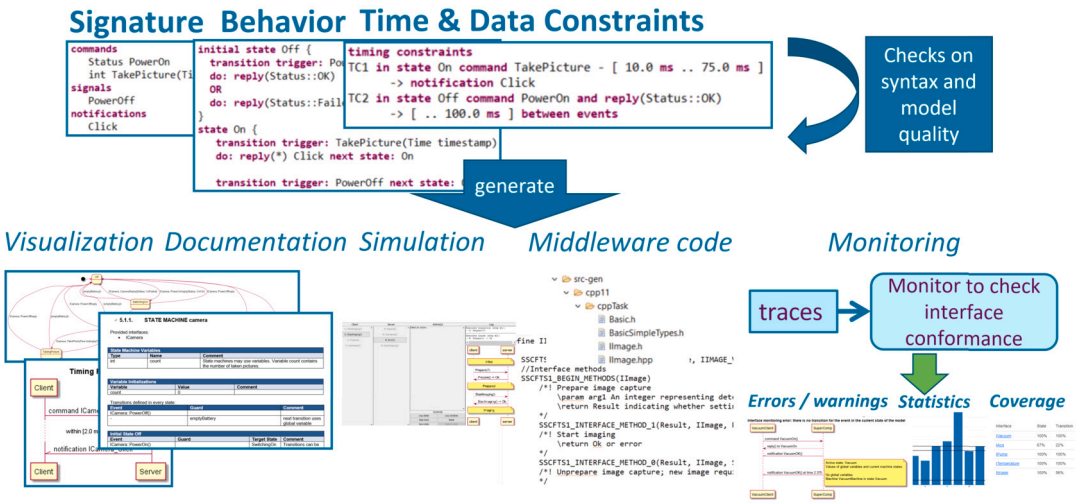

Fig. 1. Overview of ComMA: input models and main generators.

their behavior conforms to the specification. It should be noted that ComMA models are not intended for automatic generation of implementation code.

Fig. 1 shows the main ingredients of the framework. Using models as a single source, a number of generators are used to analyze the model and automate engineering tasks. The upper part of the figure indicates models that are used as input to the generators. This input comprises of an interface signature model, interface and component behavior (as state machines), and time and data constraints definitions.

The upper part of the figure also shows that there are various checks on the models. This includes syntactic checks, for instance, to detect missing reply statements, unused commands, and duplicated transitions. The more advanced quality checks take input values and values of expressions into account to detect, for example, sink states, unreachable states, and race conditions where a client call and a parallel server notification may lead to issues with unexpected calls.

The main generators, shown in the lower part of Fig. 1, address the following tasks:

- Visualization: a visual representation of models based on plantUML[2] can be generated. Behavior specifications are shown as UML state machines. Time and data constraints are represented as UML sequence diagrams.
- Documentation: creates interface documentation using a predefined MS Word template. Comments given in the models are inserted in the documents along with the generated UML diagrams.
- Simulation: simulation of a model allows obtaining early feedback and detecting errors. State machine models are transformed to executable code (in Python). Engineers use a graphical interface for a step-by-step model execution with the possibility to record the simulated trace. See Section 2.2.1 for more details.
- Interface proxy code: companies often use middleware technologies for inter-process communication. This generator creates middleware-specific proxy code from interface signatures. Currently, only the proprietary middleware of Philips is supported in a company-specific version of the tooling.
- Monitoring: the ability to monitor interfaces and components is a powerful feature of ComMA. The automatically generated monitor checks if an execution trace that contains messages observed during component execution adheres to the behavior model and the time and data constraints. The output of the monitor is conveniently shown in a dashboard that summarizes the discovered issues along with other useful diagnostic information. More details can be found in Section 2.2.2.

2.2.1. Simulation

An important facility of ComMA is the interface and component simulator. It is used by engineers to gain experience with interface and component models and allows early validation of their intended behavior.

ComMA simulators are automatically generated from interface and component models. Internally, as a first step the state machines are converted to Petri net representations based on the SNAKES Python library [4]. The Petri net representation is invisible for the user who can then explore the models by step-by-step selection of interface events. This representation is also used in the model quality checks that detect deadlocks and race conditions. The results of these checks are shown in terms of the user model as sequence diagrams that illustrate the problematic situation and its context. Fig. 2 shows the user interface of a simulator generated from a component model of a simple photo camera. The button groups for client and server show which events are enabled in the given state of the model. The sequence of the chosen events is recorded and shown as a sequence diagram.

[2] https://plantuml.com/.

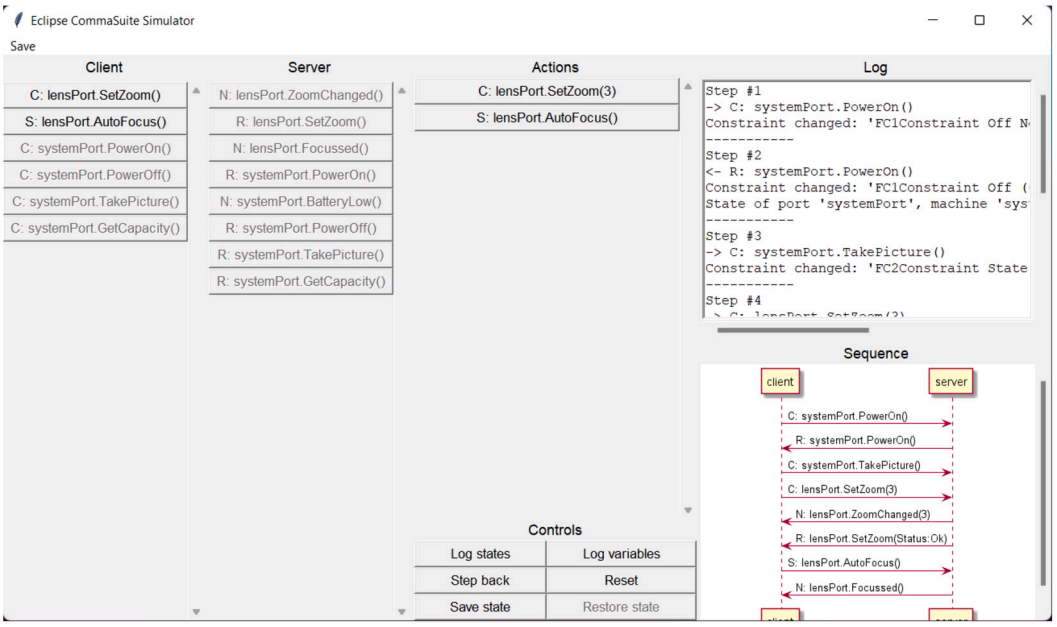

Fig. 2. ComMA simulator user interface.

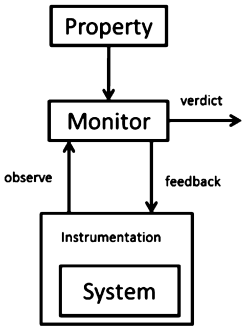

Fig. 3. General scheme of runtime monitoring.

2.2.2. *Monitoring*

Runtime verification (also known as runtime monitoring) [5,6] is an approach for checking system behavior against given properties during the system execution. The general scheme is shown in Fig. 3. The property may be given in a formal specification language (e.g., automata, logic formula, grammar), as a set of rules, or as a program. The task of the monitor is to observe the execution of the system and to produce a verdict that expresses whether the observation satisfies the properties. The observation may be a series of system states or a series of input and output events. Monitoring can be executed in a true runtime fashion (along with the system execution) or in an offline mode over a log (execution trace) with recorded observations.

Fig. 4 shows how this general approach is applied in ComMA. The interface and component behavioral models consisting of state machines, timing and data constraints play the role of properties. An executable model (implemented in Java) is automatically generated from them. Traces with system observations can be obtained in two ways: by sniffing the network traffic to and from a running component or by logging during execution. These traces are usually company-specific and reflect the used middleware or logging format. They are converted to the internal ComMA trace format and after that they can be processed by the monitor. The monitor reads the events in the trace sequentially and checks if they conform to the component and interface state machines and the specified constraints. If a violation is detected, an error is reported along with some diagnostic information, including current state of the machine, variable values, and recent events that led to the problem. The monitor also reports coverage information that includes states and transitions that were visited during monitoring of a given trace. The technical details of ComMA interface and component monitoring have already been reported in a number of publications [3,7,8].

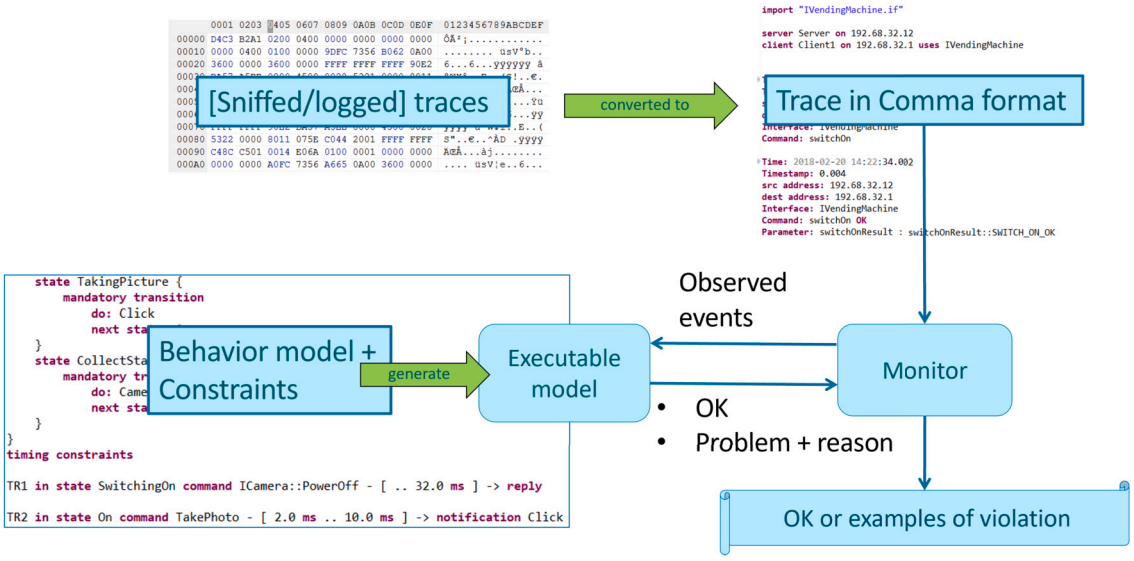

Fig. 4. Monitoring in ComMA.

2.3. Tool support

ComMA is supported by the open source tooling of the CommaSuite® project of the Eclipse Foundation.[3] The tool is available in several ways:

- As an Eclipse plugin or as a fully functional Eclipse product for Windows, Linux and macOS. The product version is used at design time when engineers define models. The models can be validated using the simulator and checked with the model quality checker.
- As a command line tool. The command line tool allows using ComMA in the continuous integration and development process. A typical scenario is an integration in the automated testing process when regression and integration tests are executed on an interface/component implementation as part of the nightly builds. During this process, execution traces are recorded as a by-product. These traces are then checked by the generated monitors.

3. Reflection on the development of the ComMA MDE approach

In this section we reflect on a number of aspects of the development and application of ComMA. The language workbench used to construct the DSL framework is described in Section 3.1. The development of the ComMA language and the generators are discussed in Sections 3.2 and 3.3. In Sections 3.4 and 3.5 we address the industrial application and adoption of the approach. A reflection on the use of MDE can be found in Section 3.6.

3.1. Language workbench

At the start of the project, the industrial users had a preference for a textual DSL to allow easy version management with operations like diff and merge, similar to what they were used to for programming languages. Since a few Philips engineers already had some experience with the Xtext language workbench [9], it was decided to use this workbench to define the language grammar. Well-formedness constraints and typing rules were implemented as Xtext validators written in Xtend [10]. Xtend is also used to implement the generators.

Our experience with Xtext as language workbench is that it is stable enough to allow creation of tools of industrial quality. Still, some tasks can be supported better if available academic results and tools are incorporated. For example, more sophisticated language reuse and composition mechanisms, and advanced parsing techniques are reported in the literature [11,12] but they have not yet found a place in the industrial workbenches.

An advantage of the Xtext/Xtend workbench is that it is relatively easy to understand for industrial engineers and they can actively contribute to the development of ComMA. For instance, Philips engineers developed a prototype task for reverse engineering, learning models from traces, and transformations between ComMA models and models of the Dezyne approach [13]. At Kulicke & Soffa a generator to the ZeroMQ middleware has been implemented.

[3] https://eclipse.dev/comma/site/download.html.

Table 1
Size indicators for the core ComMA languages showing the number of terminal and non-terminal rules in the Xtext grammars, and the number of classes and references in the metamodels. Grammars are related by grammar inheritance: Expressions inherits Types, Statements inherits Expressions, Behavior inherits Statements, etc.

Language	Terminal rules in the Xtext grammar	Non-terminal rules in the Xtext grammar	Classes in metamodel	References in metamodel
Types	22	9	21	10
Expressions	40	3	47	31
Statements	29	0	26	19
Behavior	38	0	47	76
Interface	5	0	2	1
Component	38	0	26	22

Looking back, the decision to start with a textual DSL and generate figures from it, simplified the development and allowed fast creation of prototypes in collaboration with industrial engineers. We have also considered combining visual and textual concrete syntax. In the context of the POOSL simulation language [14], colleagues combined Xtext and Sirius [15] to support a mix of textual and graphical notation [16]. This required additional development effort and maintenance was not trivial. In a recent work, Philips engineers developed a converter from state machine diagrams created with Enterprise Architect to ComMA models. Reflecting on this work, we believe that even though only a textual syntax was initially requested, the provision of multiple notations should be considered early in the project, especially if the underlying language allows it. In this respect, an interesting direction for future work is enabling blended modeling in ComMA [17].

Initially, the tooling was available as a plugin for the Eclipse IDE with non-trivial installation instructions. This was not convenient for industrial users and created also issues with managing various versions of Xtext, Eclipse, Java, etc. Later we produced a complete product that was easier to install, a solution that was much more acceptable for most users. Furthermore, the use of the Eclipse IDE was seen as a disadvantage by most Philips users, since they were used to Visual Studio as their programming IDE. We did a short investigation on the incorporation of ComMA in Visual Studio using the Language Server Protocol (LSP) [18], but this would require quite some effort and, hence, additional funding. Engineers from Thales investigated the use of ComMA in the IntelliJ IDE but concluded that there were too many disadvantages. In summary, we observe that the increase of the number of users from different organizations creates a demand for supporting multitude of notations and IDEs. We believe this is an aspect that has to be considered at an earlier stage of the project with a suitable preparation and planning.

3.2. Language development

Language engineering has been a major task in implementing the ComMA framework. The ComMA languages are developed in a modular way by defining a number of reusable grammars that are assembled by grammar inheritance in the languages used directly by the engineers (e.g., for interface and component modeling). For example, there are separate languages for types, expressions, statements, state machines and constraints.

The increasing complexity of the tool led to the need to refactor the architecture to improve maintainability. For example, the initial monolithic language has been split into a composition of DSLs. The concrete syntax of component functional constraints was changed to achieve better readability and alignment with the syntax of interface models. Currently the framework consists of more than 15 grammars for the textual concrete syntax accompanied by the corresponding metamodels defining the abstract syntax of the language.

Table 1 gives an indication of the size of the Xtext grammars and the metamodels of the core ComMA languages. It should be noted that the metamodels are automatically derived from the grammars. The Expressions, Behavior and Component languages have the biggest number of grammar rules and number of classes respectively but they mainly define concepts already known to engineers such as commonly found expressions and state machine constructs (in the Behavior language). The Interface language inherits from the Behavior language and adds a few new constructs. All languages inherit directly or indirectly from the Types language by using the Xtext grammar inheritance mechanism. The most important non-terminals are defined in Types and then reused in the inheriting languages (this is the reason for the lack of non-terminals in some languages).

One of the acceptance factors for a DSL is the pragmatics of the language. To address the language pragmatics, we tried to keep the language as simple as possible and to add new features only if they are really needed. The availability of an early implementation allowed the engineers to experiment and provide feedback about the adequacy of language concepts and the ergonomy of the concrete syntax. For instance, we discussed several representations of state machines (state based versus event based) and tried the representation of time constraints in state machines (similar to timed state machines of Uppaal [19]). Ideas for new language concepts and their syntactic representation are also frequently discussed in user meetings.

It was a challenge to keep the language as simple as possible and avoid that it becomes a general-purpose programming language thus making some tasks, such as simulation and monitoring, extremely complicated. There were several requests for nested state machines, functions, etc., often driven by programmers that were trying to express too much implementation details in ComMA. This indicates that the definition of interfaces should be a task of software architects who have a more abstract view of the roles of clients and servers; they should also have a vision on future development to construct stable interfaces. These aspects are difficult to formalize and require frequent feedback on models and coaching.

In general, backward compatibility is very important and we avoid as much as possible language changes that would require engineers to change their models. New language features are preferably defined as optional or only as syntactic sugar that would make the specifications more compact. Still, we experienced cases of changes that are not backward compatible. They were introduced at a relatively early stage of the project when the number of existing models allowed a simple manual migration. Migration of large number of models would require an automated solution, for example, by using a model transformation.

3.3. Generators

ComMA was developed in an agile way starting first with an implementation of few essential features that are sufficient for an early demonstration of the feasibility of the approach. The first version was a basic domain-specific language and a few generators with immediate practical relevance, i.e., generators for interface proxy code, visualizations, and elementary monitoring. This provided immediate benefits for the industrial projects and created interest in further applications. It also led to a stream of feature requests which have been incorporated based on user priorities. For example, inspired by the usefulness of monitoring timing constraints, users asked for monitoring of advanced data constraints. The documentation generator was also added based on requests of users. As mentioned in Section 3.1, also industrial engineers contributed to the generators.

The semantics of the ComMA language plays a key role in the generators for simulation and monitoring. Our approach to defining it is aligned with the recommendations available in the literature [20]. First, the reference semantics of the state machine language and the constraints was defined on paper and served as basis for developing the generator for monitoring. Later on, to perform model quality checks, a translational semantics was created by mapping ComMA constructs to Petri nets. As mentioned in Section 2.2.1, this semantics is implemented on top of the SNAKES library and is also used for simulation. Current work includes the use of this semantics for test generation. In fact, looking back, also the monitoring could use the Petri net semantics, but this would require an additional investment without clear benefits having in mind that the existing monitoring generator is quite stable. Still, we managed to achieve a degree of reuse by extracting a common core of a dozen of library functions shared among the generators.

The generators are one of the most complex artifacts in the framework and their testing was a major challenge. They are mainly implemented as model-to-text transformations written in Xtend. Number of issues have been identified in the literature on model transformations (see, for example [21] for an overview) that need to be addressed to obtain an efficient and effective test suite for a model transformation. The set of the input test models has to ensure a good metamodel and transformation program coverage. In addition, oracle functions are required to check whether the output of the generator is considered correct. In the context of ComMA, the output is executable code that requires its own tests. The Xtext workbench provides basic integration with JUnit but leaves the tasks of creating the input test models and oracle functions to the developer. To test the generators we use a set of manually created input models that are transformed to executable code (in Python or Java) which in turn is tested. For example, the implementation of a monitor (in Java) obtained from a given interface model is tested against a set of predefined traces for which the result of monitoring is known.

We explored approaches that might automate some of the testing tasks. The generation of test input models using partial models and constraints can be automated with tools like USE and EFinder [22]. Furthermore, ComMA state machines can be expressed in the language of model-based testing tools that can help in addressing the challenge for creating tests and test oracles for the generated code. The results of such an exploration are reported in [23].

3.4. Industrial application

ComMA is applied to model and develop a number of software interfaces as part of the production software of Philips IGT. We have observed the following application scenarios and the perceived benefits.

- Modeling and monitoring of an existing interface that is already implemented. In this scenario the implementation of the interface and the client software are already available. Typically, interface documentation exists in natural text and diagrams. This scenario illustrates the original motivation for ComMA: interface definitions may be ambiguous, incomplete, and cannot be readily used for automatic analysis. Interface modeling stimulates engineers to think of time and data constraints that might have been left implicit. Monitoring is performed on execution traces obtained from the available tests. Monitor failure is due to an incorrect implementation or an incorrect model (we have experienced both situations).
- Modeling of a new interface that is developed in-house. In this scenario, the interface is defined and then validated using the simulator. Model quality checks are applied to analyze the model for deadlocks, race conditions and unreachable states. Once the engineers are satisfied with the model, the interface can be implemented. The application of ComMA then proceeds as in the first scenario. In addition, the interface documentation is automatically generated using the model.
- Modeling of interfaces of a third-party component for which client software needs to be developed. In this case the client software and the model are developed iteratively based on the provided documentation. The process of modeling helps in identifying and resolving ambiguities and incomplete information. Monitoring is used to check if the provided implementation conforms to the specification (captured in the model). In some cases, the timing requirements of the interface are unclear. The expected (or hypothesized) timing behavior can be specified as constraints and the generated statistical information from the monitoring is used to get insight on the actual behavior. This scenario has been applied in a number of cases and helped in clarifying the interface and identifying conformance violations.

  ComMA was used to model the interfaces of a number of recently introduced third-party components. Engineers were able to

find multiple issues early in the development process. These issues would have been difficult to debug in the field, or might have introduced unwanted behavior in future upgrade scenarios.

3.5. Industrial adoption

Developing and applying a model-based method and tool in a large organization is a challenging task. The existing way of working is impacted because new tools, artifacts and tasks are included in the existing development process. We recognize several factors that influenced the successful adoption of the ComMA framework. These are: long term commitment and support from management, ensuring acceptance from engineers, fast response to maintenance requests (fixing defects and adding new features), stability of the tool, and personnel training.

Strong support from management and early adoption from several enthusiastic engineers were key factors in the successful deployment of ComMA. In our experience, both are essential and need to be secured in the beginning of the project. Observe that the application of ComMA requires an initial investment by component developers to define interfaces in more detail and check conformance of implementations with respect to these interfaces. The benefits will be visible much later in the development process, e.g., by system testers, when components are integrated. From that perspective, it is essential to have support from the management who is responsible for the efficiency of the development process as a whole. In addition, adoption is further facilitated if component engineers also have direct benefits, such as a convenient way to generate middleware code and a fast way to obtain documentation according to the most recent guidelines.

At Philips IGT, management did an analysis of integration issues and formulated a strong business case for the improvement of interface definitions and checks. It should also be noted that the management had a positive opinion on model-based approaches as a way to reduce maintenance costs and increase productivity via automation. They also pushed the use of ComMA and made it mandatory for all interfaces that are used across subsystems. The company software development handbook mandates that these interfaces are modeled and interface middleware proxy code is generated. The interface conformance is checked against the ComMA specifications as part of the continuous integration pipeline when executing the automated test scenarios.

Training was delivered in the form of workshops attended by engineers and managers. This is a continuous effort that spans several years. New ComMA users attend a single four-hour long workshop that includes a short intro and a demo followed by a hands-on session. The intro outlines the basic principles of MDE (models as main artefacts and single source of truth) and the role of DSLs. During the hands-on, the participants perform activities for modeling an interface, generating artefacts, and monitoring. This is usually sufficient for the engineers to get started and complete activities from a more extensive tutorial on their own. The workshops also contributed to creating awareness in the organization about the capabilities of the tool and its benefits. Significant effort was spent on creating both basic and advanced demonstration examples, detailed user guides, tutorials, and help functionality. The development team reacted fast to user requests and team members gave direct support to engineers when performing the modeling tasks (as a part of the industry-as-lab approach).

The ComMA framework addresses the general problem of interface and component modeling and as such is not industry or company specific. Efforts were made to attract interest in a wider group of companies that may benefit from the method and the tools. This process of community building became even more relevant after ComMA was released as an open source project in April 2021. A few actions in this direction are worth mentioning.

The experience from applying ComMA was shared in a number of popular magazine articles [24,25] aimed at a broader audience from the high-tech industry. These publications usually lead to requests for further information.

A ComMA user group was established with the purpose of sharing industrial experiences, presenting new features and identifying common needs. This helped the development team to identify a core set of tool features that are generic and reusable and to define a strategy for developing company-specific extensions and adaptations. An example of such an extension is the result of a project with Thales Nederland on the adaptation of software interfaces and model quality checks using the ComMA framework. These checks lead to the Petri net semantics of the languages [25,26]. Also interesting is that Thales trained two of their own engineers such that they can teach their colleagues a methodology based on ComMA [27]. More generally, reflecting on the specific company extensions, we observe that they mainly address generators for the middleware technologies used in a company, support for company-specific log and trace formats and documentation templates. To facilitate the implementation of such extensions, a detailed development guide was provided to the interested parties.

3.6. Reflection on MDE

In this section we reflect on the choice of using MDE for modeling component interfaces and on our experience in applying this approach.

Why MDE helped

At a global level, MDE promotes usage of models as primary artifacts for developing and maintaining software systems. When defined at the right level of abstraction and in a suitable language, models are usually more compact, and easier to analyze than implementations in general purpose programming languages. Finally, the use of generative techniques (e.g., model transformations) allows automation in development tasks.

We observed that the current way of specifying component interfaces in industrial context often relies on specific implementation technologies (e.g., Microsoft IDL) and informal, sometimes incomplete, textual and diagrammatic descriptions of interface behavior and timing requirements. This hinders the migration to new technologies and the automation of checks if the implementation conforms to the specifications. Following the MDE principles, ComMA treats interface and component models as primary artifacts and a single source of truth for a number of engineering tasks. The usage of models and DSLs allows more explicit, precise and machine processable specification of interfaces, a feature that is not present in a document-based approach. Furthermore, the ComMA approach provides a degree of technology independence: abstract interface and component models can easily be mapped to a particular component or middleware technology and this process can be repeated when a new technology emerges.

Many applications of MDE use models to automatically derive software implementations. ComMA does not immediately fall in this category. Still, the framework provides automatic generation of many artifacts, mainly aimed at supporting validation and verification by means of, for instance, simulators and monitors. Implementing such tools manually is a time consuming and error-prone process. Philips also expects an increase in efficiency, as reported in [28].

Challenges

Often new MDE tooling starts as a prototype to investigate whether the ideas are useful in practice. Our experience is that it is important to avoid ad-hoc development of such prototypes. When users react positively to the approach they usually expect professional tool support quickly. To keep the momentum, there is no time for a fresh start. Hence it is important to use a professional setup from the start, including good version management and a continuous integration and delivery (CI/CD) pipeline,

Such a professional setup is also important for the long term maintenance of a successful MDE approach. For ComMA, the idea was that maintenance would be easier to fund with a larger user community. This, however, was blocked initially by issues with intellectual property rights. When this was resolved, the Philips-specific tool was turned into an open source tool with the financial support of several industrial partners. The Philips version is derived from the open source version by adding a few Philips-specific aspects, such as a generator for Philips-specific middleware and a Philips document template. Still it is difficult to find industrial users as committers for the open source project.

Another aspect related to long term maintenance of the tool is the fact that for the most of the intended industrial users, development and maintenance of MDE tools is not part of their core business and expertise. It is expected that if the project brings value to users, the tasks of maintenance, consultancy and training are handled by a provider of engineering services with a suitable expertise while sharing the cost among the users. Related to this is another challenge we recognize: forming a good development team with MDE expertise that can ensure continuity of the activities. The ComMA development team includes core members with scientific and practical expertise in MDE and language engineering. New team members are encouraged to first gain experience as ComMA users and then to understand and further develop the tool itself. They usually have basic MDE knowledge and are supported by mentoring and frequent feedback.

Using MDE in other Projects

We would apply an MDE approach again in new projects taking into account the experience gained from this one. More concretely, the following aspects should be considered:

- Deciding between using and tailoring existing tools and languages versus developing a new language from scratch.
- Connection to existing MDE tools when certain desired functionality is not available. For example, we recently investigated together with an academic partner the connection to Capella in the context of the alignment between model-driven systems engineering and software engineering processes [29]. This work revealed that the two processes are intertwined, use shared or related artefacts, while still using different tools that need to be aligned.
- Focus earlier on a common underlying semantic model that is transparent to the user (so that it can evolve and be changed) and that serves as foundation for all the analysis tasks. Currently in ComMA, the simulation and the model quality checks share the Petri net representation, but monitoring is not directly based on it.
- Consider from the beginning of the project support for multiple notations and development environments to anticipate users with different preferences.
- Start open source development early and arrange the related intellectual property aspects.

4. Related work

The topics of specifying software interfaces and architectures based on components have been extensively studied since the late 1990s. A number of approaches and languages have been proposed with different goals and underlying formalisation. The seminal work on interface automata gives a foundation for formalizing interface descriptions [30]. The specification of the behavior of interfaces has been addressed both at the level of models [31] and programming languages [32,33]. Architecture description languages based on the component-port-connector style have been proposed both by the academia and industry. Examples of such works are AADL [34], MontiArc [35], BIP [36] to name only few of them. Some of these languages have been equipped with formal semantics to support various forms of analysis (see for example, two formalizations of AADL [37,38]).

The concepts of provided and required interfaces, ports, and connectors are well-known from modeling languages such as UML [39] and its extension for systems modeling SysML [40]. In UML, interfaces may own a protocol state machine which is similar to the ComMA state machines. To specify timing properties, Marte [41] can be used, a UML profile for modeling real-time systems.

In general, UML is often tailored to a particular problem area by means of profiles, which can be seen as a domain-specific extension of the language. Analysis of UML models requires dedicated tools and a choice of a suitable formalization of the language semantics [42]. This is not trivial for such a rich language, whereas engineers only need a subset of the language. To obtain a dedicated modeling framework for software components and their interfaces, ComMA has been developed using a set of standalone DSLs and tailored generators for these DSLs. It was a conscious choice to use the same concepts as UML, since they are familiar to practitioners. Furthermore, we reuse many concepts from the existing architectural description, software modeling and also time-based logic languages (for the purpose of defining timing constraints). From that perspective, ComMA can be perceived as an integration of known concepts and techniques carefully selected on the basis of user needs and aiming at easy application in industry.

Another dedicated DSL for interface specification is also used by Verum's ASD [43] and its successor Dezyne [13]. Dezyne provides tooling for component interface modeling and design, allowing formal checks that a design model conforms to an interface specification. To avoid state explosion of the underlying model checker, the interface modeling language is less expressive than ComMA. For instance, control decisions must be data independent and the specification of time behavior is not supported. ComMA uses runtime verification techniques instead of an underlying model checker, thus allowing more expressive specifications with decisions based on data.

The literature on runtime verification contains a large number of languages to specify properties, see for instance the taxonomy of [44]. Specification can be described operationally, e.g., as a finite state automaton, or more declarative, e.g., as a temporal logic formula. In contrast to languages directly based in formal logic, Dwyer et al. [45] identified patterns for properties observed in industrial practice. In ComMA, the time and data constraints are based on such patterns to obtain a concise and intuitive syntax. To express component functional constraints, we have chosen the state-based specifications which are already used in interfaces since users appreciate this operational style. A related runtime verification approach for component-based system is proposed by Falcone et al. [46]. Similarly to ComMA, they use state machines; in their approach ports accept simple values, whereas in our approach ports are associated to interfaces with signatures that may use complex data structures.

5. Conclusion and future work

We presented the ComMA framework: a family of domain-specific languages and tools that support model-based engineering of components with an emphasis on precise and explicit specification of component interfaces. The framework integrates techniques and results from different research areas and provides a single entry point for engineers to specify and develop components based on the MDE principles.

Despite the overall positive experience in applying ComMA in industry, we also observed several challenges. In general, interface and component modeling is not an easy task, it requires proper abstraction from the implementation details. Engineers that perform this task often need training, supervision and feedback to be provided in an organized way. Not all new tool features are adopted easily and quickly, this is a process that needs a stimulation and has to be approached systematically, for example, by providing training and workshops. Performing a systematic usability study based on the engineers' experience with ComMA can shed light on still unidentified points for improvement and potential new features. This is an interesting direction for future work.

Currently, the status of languages, the simulator and the monitoring functionality can be considered stable, still occasionally facing maintenance requests. The ongoing and future work is focused on support for efficient test generation (also based on the Petri net semantics) for components with provided and required interfaces. This support will be tested on large and complex interfaces. Furthermore, together with the Philips MR unit we investigate the application of ComMA to software/hardware interfaces and the analysis of timing behavior. In collaboration with the Eindhoven University of Technology we also investigate a connection to a model checker to check the consistency of models extending the current model quality checks.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The research is carried out as part of the "Model-Based Testing with ComMA" program under the responsibility of TNO-ESI in cooperation with Philips. The research activities are supported by the Netherlands Ministry of Economic Affairs and Climate, and TKI-HTSM/23.0181. Our thanks goes to Wytse Oortwijn for his valuable feedback on the manuscript and to the anonymous reviewers for their constructive suggestions that helped greatly to improve the structure and the content of the paper.

References

[1] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012.

[2] C. Potts, Software-engineering research revisited, IEEE Softw. 19 (9) (1993) 19–28.
[3] I. Kurtev, J. Hooman, M. Schuts, Runtime monitoring based on interface specifications, in: ModelEd, TestEd, TrustEd, Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday, in: LNCS, vol. 10500, Springer International Publishing, 2017, pp. 335–356.
[4] F. Pommereau, SNAKES: a flexible high-level Petri nets library (tool paper), in: R.R. Devillers, A. Valmari (Eds.), Application and Theory of Petri Nets and Concurrency - 36th International Conference, PETRI NETS 2015, Brussels, Belgium, June 21-26, 2015, Proceedings, in: Lecture Notes in Computer Science, vol. 9115, Springer, 2015, pp. 254–265.
[5] Y. Falcone, K. Havelund, G. Reger, A tutorial on runtime verification, in: M. Broy, D.A. Peled, G. Kalus (Eds.), Engineering Dependable Software Systems, in: NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, IOS Press, 2013, pp. 141–175.
[6] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebraic Program. 78 (5) (2009) 293–303, https://doi.org/10.1016/j.jlap.2008.08.004.
[7] I. Kurtev, M. Schuts, J. Hooman, D.-J. Swagerman, Integrating interface modeling and analysis in an industrial setting, in: Proc. 5th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD 2017), 2017, pp. 345–352.
[8] I. Kurtev, J. Hooman, Runtime verification of compound components with ComMA, in: N. Jansen, M. Stoelinga, P. van den Bos (Eds.), A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday, in: Lecture Notes in Computer Science, vol. 13560, Springer, 2022, pp. 382–402.
[9] Xtext, http://www.eclipse.org/Xtext/, 2015.
[10] Xtend, http://www.eclipse.org/xtend/, 2015.
[11] A. Butting, R. Eikermann, O. Kautz, B. Rumpe, A. Wortmann, Modeling language variability with reusable language components, in: T. Berger, P. Borba, G. Botterweck, T. Männistö, D. Benavides, S. Nadi, T. Kehrer, R. Rabiser, C. Elsner, M. Mukelabai (Eds.), Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018, ACM, 2018, pp. 65–75.
[12] A. Afroozeh, J. Bach, M. van den Brand, A. Johnstone, M. Manders, P. Moreau, E. Scott, Island grammar-based parsing using GLL and Tom, in: K. Czarnecki, G. Hedin (Eds.), Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 7745, Springer, 2012, pp. 224–243.
[13] Dezyne, https://www.verum.com/, 2022.
[14] POOSL, early validation by simulation, https://www.poosl.org/, 2023.
[15] Sirius, https://eclipse.dev/sirius/, 2023.
[16] K. Staal, A. Mooij, Integrating textual and graphical editing, https://www.slideshare.net/Obeo_corp/siriuscon2016-integrating-textual-and-graphical-editing-in-the-poosl-ide, 2016.
[17] F. Ciccozzi, M. Tichy, H. Vangheluwe, D. Weyns, Blended modelling - what, why and how, in: L. Burgueño, A. Pretschner, S. Voss, M. Chaudron, J. Kienzle, M. Völter, S. Gérard, M. Zahedi, E. Bousse, A. Rensink, F. Polack, G. Engels, G. Kappel (Eds.), 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019, IEEE, 2019, pp. 425–430.
[18] Language server protocol (LSP), https://microsoft.github.io/language-server-protocol/, 2023.
[19] UP4ALL, Uppsala, Uppaal, http://www.uppaal.com/, 2015.
[20] B. Combemale, X. Crégut, P. Garoche, X. Thirioux, Essay on semantics definition in MDE - an instrumented approach for model verification, J. Softw. 4 (9) (2009) 943–958, https://doi.org/10.4304/jsw.4.9.943-958.
[21] B. Baudry, S. Ghosh, F. Fleurey, R.B. France, Y.L. Traon, J. Mottu, Barriers to systematic model transformation testing, Commun. ACM 53 (6) (2010) 139–143, https://doi.org/10.1145/1743546.1743583.
[22] J.S. Cuadrado, M. Gogolla, Model finding in the EMF ecosystem, J. Object Technol. 19 (2) (2020) 10:1–21, https://doi.org/10.5381/jot.2020.19.2.a10.
[23] T.P. Hoeijmakers, Testing an industrial code generator with model-based testing, Master's thesis, Eindhoven University of Technology, 2022, https://research.tue.nl/en/studentTheses/testing-an-industrial-code-generator-with-model-based-testing.
[24] M. Schuts, D.-J. Swagerman, I. Kurtev, J. Hooman, Improving interface specifications with ComMA, Bits & Chips (14 September 2017), https://bits-chips.nl/artikel/improving-interface-specifications-with-comma/.
[25] N. Roos, ComMA interfaces open the door to reliable high-tech systems, Bits & Chips (8 September 2020), https://bits-chips.nl/artikel/comma-interfaces-open-the-door-to-reliable-high-tech-systems/.
[26] B. Hilbrands, D. Bera, B. Akesson, Partial specifications of component-based systems using Petri nets, in: M. Köhler-Bussmeier, D. Moldt, H. Rölke (Eds.), Petri Nets and Software Engineering 2022 Co-Located with the 43rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2022), Bergen, Norway, June 20th, 2022, in: CEUR Workshop Proceedings, vol. 3170, CEUR-WS.org, 2022, pp. 21–39, https://ceur-ws.org/Vol-3170/paper2.pdf.
[27] B. Akesson, Model-based specification, verification, and adaptation of software interfaces - the DYNAMICS approach, https://esi.nl/news/blogs/the-dynamics-approach, 2022.
[28] Codemotion, Domain-specific languages could scale up your code with ComMA, https://www.codemotion.com/magazine/languages/domain-specific-languages-could-scale-up-your-code-with-comma/, 2019.
[29] D. Hendriks, Exploring conceptual alignments for interface specification in model-driven systems- and software engineering processes by analyzing Capella and ComMA, Master's thesis, Eindhoven University of Technology, 2022, https://research.tue.nl/en/studentTheses/exploring-conceptual-alignments-for-interface-specification-in-mo.
[30] L. de Alfaro, T.A. Henzinger, Interface automata, in: A.M. Tjoa, V. Gruhn (Eds.), Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001, ACM, 2001, pp. 109–120.
[31] K. Kähkönen, J. Lampinen, K. Heljanko, I. Niemelä, The LIME interface specification language and runtime monitoring tool, in: S. Bensalem, D.A. Peled (Eds.), Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers, in: Lecture Notes in Computer Science, vol. 5779, Springer, 2009, pp. 93–100.
[32] G. Leavens, Y. Cheon, Design by Contract with JML, 2003.
[33] J. Hatcliff, G.T. Leavens, K.R.M. Leino, P. Müller, M.J. Parkinson, Behavioral interface specification languages, ACM Comput. Surv. 44 (3) (2012) 16:1–16:58, https://doi.org/10.1145/2187671.2187678.
[34] SAE AS5506A, Architecture Analysis & Design Language (AADL), 2009.
[35] A. Haber, Montiarc - architectural modeling and simulation of interactive distributed systems, Ph.D. thesis, RWTH Aachen University, Germany, 2016, http://www.shaker.de/de/content/catalogue/index.asp?ISBN=978-3-8440-4697-7.
[36] A. Basu, S. Bensalem, M. Bozga, P. Bourgos, J. Sifakis, Rigorous system design: the BIP approach, in: Z. Kotásek, J. Bouda, I. Cerná, L. Sekanina, T. Vojnar, D. Antos (Eds.), Mathematical and Engineering Methods in Computer Science - 7th International Doctoral Workshop, MEMICS 2011, Lednice, Czech Republic, October 14–16, 2011, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 7119, Springer, 2011, pp. 1–19.
[37] B.R. Larson, P. Chalin, J. Hatcliff, Bless: formal specification and verification of behaviors for embedded systems with software, in: G. Brat, N. Rungta, A. Venet (Eds.), NASA Formal Methods, Springer Berlin Heidelberg, 2013, pp. 276–290.
[38] P.C. Ölveczky, A. Boronat, J. Meseguer, Formal semantics and analysis of behavioral AADL models in real-time maude, in: J. Hatcliff, E. Zucca (Eds.), Formal Techniques for Distributed Systems, Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, the Netherlands, June 7-9, 2010. Proceedings, in: Lecture Notes in Computer Science, vol. 6117, Springer, 2010, pp. 47–62.
[39] Unified Modeling Language (UML), http://www.uml.org/, 2017.

[40] Systems Modeling Language (SysML), https://www.omgsysml.org/, 2017.
[41] UML profile for MARTE (Modeling and Analysis of Real-Time and Embedded systems), https://www.omg.org/spec/MARTE, 2019.
[42] H. Kim, D. Fried, P. Menegay, G. Soremekun, C. Oster, Application of integrated modeling and analysis to development of complex systems, Proc. Comput. Sci. 16 (2013) 98–107.
[43] G.H. Broadfoot, ASD case notes: costs and benefits of applying formal methods to industrial control software, in: J.S. Fitzgerald, I.J. Hayes, A. Tarlecki (Eds.), FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3582, Springer, 2005, pp. 548–551.
[44] Y. Falcone, S. Krstić, G. Reger, D. Trayel, A taxonomy for classifying runtime verification tools, Int. J. Softw. Tools Technol. Transf. 23 (2021) 255–284.
[45] M.B. Dwyer, G.S. Avrunin, J.C. Corbett, Patterns in property specifications for finite-state verification, in: B.W. Boehm, D. Garlan, J. Kramer (Eds.), Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999, ACM, 1999, pp. 411–420.
[46] Y. Falcone, M. Jaber, T. Nguyen, M. Bozga, S. Bensalem, Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation, Softw. Syst. Model. 14 (1) (2015) 173–199, https://doi.org/10.1007/s10270-013-0323-y.