I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, C. Seidl (Hrsg.): Modellierung 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 245

Systematic Refinement of CPS Requirements using SysML, Template Language and Contracts

Markus Grabowski [1], Bernhard Kaiser [2], Yu Bai [3]

Abstract: In these days, we encounter the transition from traditional closed and restricted-purpose embedded systems towards networked Cyber-Physical Systems. This applies to many industries, but in particular to the automotive industry, where assistance and automated driving functions are shaped out of complex combinations of functions and electronic control units, and even the car as a whole becomes part of a larger network of many vehicles plus infrastructure. Still, verifiable assertions must be available in the end to satisfy the safety case. The specification skills in industry often turn out to be insufficient. Even today, the mandatory V-model is hard to apply in practice and expressing appropriate requirements and refinements along with the evolution of the architecture is a hard thing to do. When development becomes agile and centered around component reuse, things become even more complex. We report about our experience with the application of contract-based development and explain keystones of our approach. We present a new template language called SSPL that allows the specification of requirements and assertions on every system architecture level and show how contract-based requirements refinement can go hand in hand with architecture refinement in SysML. We further present our Eclipse-based tool SAVONA that enables practical application of the approach.

Keywords: contract-based design; template language; system refinement; system verification; cyber physical systems

1 Introduction

Almost all technology-related industries are facing a rapid transition from formerly closed, local, restricted-purpose Embedded Systems to open, interconnected and jointly acting Cyber Physical Systems (CPS), uniting different physical domains with IT and networking technology. This affects existing industry branches, such as automotive and industrial automation, but also enables entirely new application fields, such as home automation, sensor networks, and the Internet of things. This transition can exemplary be observed by the automotive industry which is forced into the most dramatic transformation since the invention of the car. Novel assistance functions have arisen, so that in many cases, sensors and actuators made in a variety of technologies serve for many different functions. Some of these components have been designed and released years before the functions they are later used in. Consequently, the original developers of these components had

[1] Assystem Germany GmbH, Berlin, Germany, mgrabowski@assystem.com
[2] Assystem Germany GmbH, Berlin, Germany, bkaiser@assystem.com
[3] Assystem Germany GmbH, Berlin, Germany, ybai@assystem.com

no chances to overlook all future implications, and the architect wanting to reuse an existing component today does not know under which assumptions it had been developed. Still, systematic properties like correct behavior and safety have to be validated in the end. Highly automated driving (HAD) not only accelerates this trend, but also enforces open interconnections between vehicles and their environment (V2X), because the reach of on-board sensors is limited and therefore external information about road conditions, incidents etc. must be received and directly influences safety-relevant maneuver decisions. The single car is no longer the biggest conceivable system scope, but the whole traffic system may act as a System-of-Systems (SoS). As new suppliers and off-board services enter the automotive domain, technologies and development approaches from domains like IT are harshly contrasting the traditional automotive processes like the V-model. In the light of these tremendous changes, it can be expected that automotive industries will have to adapt their way of specifying, developing, and releasing their CPSs towards more agile and reuse/recombination-oriented approaches. It can be questioned if the traditional V-model will carry the industry into the future at all, but even in places where the V-model is officially mandatory, we have observed many flaws in its application in industrial practice. Foremost, these flaws affect the first and most important process phases, the requirements engineering and architecture specification. Requirements found in practice are often imprecise or badly put in words, although the use of template languages has been encouraged for many years [HJD04]. Specifications from car OEMs to suppliers often include many details about process aspects like manufacturing, environmental tests, standards compliance etc. but when it comes to the core function of the component (e.g. steering controller, radar sensor), there is not much of the component's behavior specified in detail. Additionally, when Safety Integrity Levels (ASILs) are assigned, it is sometimes not clear which property or service exactly is subject to the ASIL. Moreover, requirements are often stated on a wrong scope. For instance, a requirement towards a radar sensor, specifying the initialization of a braking action when detecting certain obstacles, is clearly a requirement on vehicle scope, but not on sensor scope; nevertheless, such kinds of requirements are often found in specifications towards the radar sensor supplier. On supplier level, requirements on system level are simply passed through to software component developers, without performing the architect's core duty: to decompose the requirement into sub-requirements for each component and verify that the whole shows the expected behavior. Requirement specifications are sometimes only exchanged at project setup and not further kept up-to-date so that their usefulness as a reference for verification and safety case creation can be doubted. Backward requirements from suppliers to the OEM or assumptions about the usage environment are usually not formally captured. Flaws in implementation or verification are often the consequence, in the worst case leading to actual safety risks. Contract-based development approaches have become more and more mature and popular in the last couple of years and actually have the potential to support better requirements refinement, in particular when combined with model-based development. Template-based assertion and contract specification languages brought up by recent research projects [CE10, Bo15] have contributed a lot, but still need to be extended to allow expressing all necessary kinds of requirements on system and software level.

In this paper, we present an advanced template language that goes another step in transferring contract-based development approaches into industrial practice, allowing the stepwise requirements refinement to go hand in hand with model-based system design. After collecting some requirements regarding a modern specification process that have guided us when developing our approach, we will briefly give a short introduction on contract-based design and template languages, as those are the key ingredients of our methodology. After that, we present our *System Specification Pattern Language (SSPL)* with which behavioral system properties can be described in a well readable and unambiguous way. Furthermore, we describe the method of utilizing contract-based design and SSPL in a stepwise system refinement process [Ka15]. Additionally, we shortly introduce our tool framework SAVONA, which implements the method and offers support for creating system models and template expressions and allows early system verification. At the end, we report on a first experience with our approach applied on a research case study.

2 Requirements towards an approach to address these issues

Based on our own industrial project experience, we have collected some requirements for a specification method that could help industrial companies mastering the aforementioned challenges. The approach should not be restricted on a Waterfall or V-Model, but also allow going bottom-up, i.e. assembling a new system from preexisting function blocks or practicing agile development methods. It should address the system level and multi-physics domains, thereby be based on industry-accepted standard languages such as SysML, Simulink, Modelica, AADL, and the like. This implies interfacing to standard tools and supporting co-working at different sites on black-box-level, which helps protecting intellectual property. Tools exploiting the specification technique should provide side-by-side development of architecture and requirements, thereby allowing the specification of requirements on every level of the architecture. A fixed order of requirements formulation and architecture modeling should not be imposed. All requirements or assertions shall be bound to architectural components and formulated on their scope by only referring to externally observable behavior at the component's ports without making any assumptions about the internal design and implementation. A central point is the guided and verifiable refinement of the requirements of the super-component onto the requirements of its sub-components. Regarding the specification language for requirements or properties, the key requirement is sufficient expressive power for most kinds of embedded systems and CPS, not stopping at standard phrases like 'The System shall <perform action>'. A highly extensible and adaptable language is needed which still has a precisely defined syntax in Backus-Naur-Form (BNF) or similar to enable automated parsing or bottom-up construction using a template editor. If architecture development and requirements specification go hand in hand, the architecture model including class hierarchy is a natural source for an ontology of terms to be used. To assure correct and safe operation of the highly integrated system and to assure safe reuse of components in the future, it is necessary to capture not only the requirements

towards a component, but also the assumptions towards the operational environment under which the component has been developed.

One approach that seems specifically suited to match these requirements is contract-based development (CBD), that will be at the core of the approach presented in this paper and will be explained in more details below. However, with its origins in formal software specification, existing approaches to CBD do not fulfill all of the listed requirements. We will complement CBD with approaches to specify assertions by templates that are close to natural English languages for better understandability. We will also tightly integrate it with architecture modeling, centered about SysML Internal Block Diagrams (IBD) as the modeling technique for the static system and software architecture on all levels before more specific modeling techniques like Simulink, UML, or hardware block diagrams take over.

3 Fundamentals and related work

In this section, we give a short introduction to contracts and contract-based design. Additionally, specification languages for requirements and contracts are discussed. Since these topics are quite extensive we will also refer to some fundamentals and only shortly outline related work.

3.1 Contract-based design

Initially intended as a verification method for sequential software, Bertrand Meyer [Me92] introduced Contracts using preconditions (which must hold at program entry), post-conditions (which must hold at program exit), and invariants (which must hold at every point in time). This idea has been adopted later to component-based software and system development. A system applying the *'contract-based' design* is represented by a system model containing components, ports and signals. The system itself is seen as a hierarchical composition of components, which can exchange information, energy or mass flow through their interfaces, called ports, that are connected to each other via signals. In the contract-based design paradigm, contracts are defined using assertions that allow black box specifications of components, which means while describing inputs and observable behavior from the outside, the inner (structure and) working remains unknown. Those assertions need to be distinguished between *assumptions* about conditions of their environment and the *guarantees* that can be provided given that the assumptions are fulfilled. This separation into assumptions and guarantees allows arguing about the functioning of a component composition, as for every contract can be verified whether the assumptions are met by the guarantees.
Applying contracts on a system and its components can lower the complexity of verifying the implementation against the specification. Fig. 1 shows an example of a contract specification for an airbag system. The system's contract specifies that the airbag expects a given value

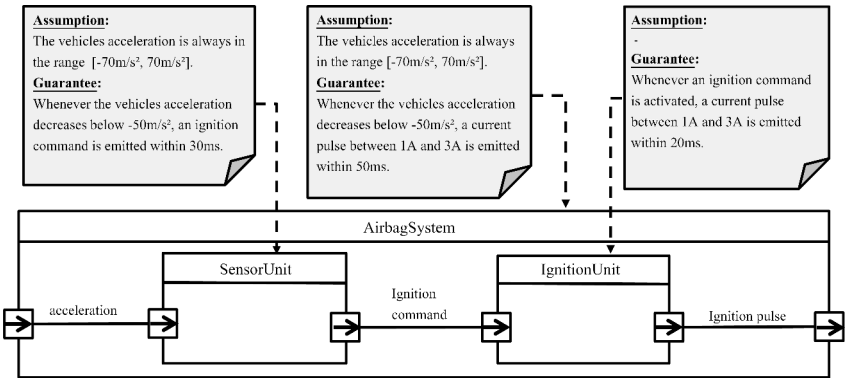

Fig. 1: Example of a contract specification

range on the input port *acceleration*. If that assumption is fulfilled the airbag guarantees that whenever the value decreases below a certain value it sends a current pulse within 50ms. The system is decomposed into two subsystems each having one contract. By assuming that the functionality on subsystem *IgnitionUnit* depends on the output of subsystem *sensorUnit* and that the subsystems would not be annotated with contracts, validating the contract of the overall *AirbagSystem* would be very complicated, as the composed behavior of both subsystems has to be computed, leading to large state spaces. By using contracts for the two subsystems, we can validate the sub-contracts locally and avoid the complex state machine composition [Ci12].

3.2 Template languages

Because formal expressions are hard to write and understand by non-experts, there is a huge suppression in using them. That turns out to be very unfortunate as they provide many striving characteristics, with which requirements engineering processes would benefit from. A well-defined syntax and semantics offer only one way to interpret statements, making things like automatic validation, tracing etc. possible. Expressions in natural language might be easier to read, but they have no constrains in syntax and semantics resulting in ambiguous statements which make automation nearly impossible without further work. In addition, they will likely always need a person with an appropriate domain knowledge to interpret and validate the expressions correctly.
Template Languages can close the gap between purely formal and unconstrained natural language. They provide a well chosen set of allowed sentence patterns, which results in a straight syntax to unify expressions making it easier to read for the recipient. Ideally, the template language also has unambiguous semantics, leaving only one way to interpret an expression. As an example, Hull et al. [HJD04] proposed to use *Requirement Boilerplates* like '*The <system> shall <function> <object> every <performance> <units>*.', where

<*keyword*> are placeholders to be replaced by the requirement engineer. Similar approaches have been done by [IKD09, Ma09, DSS12]. Having an expression written in a specified syntax and semantics, a machine, which has knowledge about the used grammar, can now parse and process it further allowing automatic verification. With this, template languages have the same advantages as formal languages. In addition, they feature a better readability as they are only constraining natural language instead of expecting formal expressions. The only drawback is the overhead of assigning keywords a meaning, so that a machine can interpret them correctly.

4 System Specification Pattern Language (SSPL)

In a previous work [Gr17], we analyzed existing template languages to find out why those are not currently used in practice yet. We focused on the semi-formal languages *Goal and Contract Specification Language* (GCSL)[Bo15], *Requirement Specification Language* (RSL)[CE10] and the *Property Specification Pattern* (PSP)[Au15] as these are able to describe system behavior and have influence on research and/or industry. Other than that, they featured a well-defined semantics in combination with the possibility of being translatable to a verifiable formal language. Language attributes like readability and expressiveness were evaluated by expressing real industrial system requirements with the respective languages. Others like formalization capabilities have been evaluated by the completeness of syntactic and semantic definition. We found that none of the languages features a highly readable and expressible syntax while providing unambiguous (and formal) semantics. That gave us reason to come up with an own template approach called *System Specification Pattern Language* (SSPL), which improves on the deficits of existing template languages.

4.1 Language Characteristics

To represent an applicable specification language, SSPL has been designed to feature **highly readable and well understandable expressions**. We ran a study [Gr17] on the acceptance of our template expressions from a readers' perspective and showed that SSPL performed strictly better than other template languages and is even received as less ambiguous than natural language.
SSPL enables the specification of **simple and complex system behavior** by allowing chains of basic expressions. We were able to achieve an overall translation rate of about 92%[Gr17] of all functional requirements from an automotive light system specification provided by Daimler within the project ASSUME[4]. A full description of the language as well as its syntax in Backus-Naur-Form and semantics in temporal logic can be found in [Gr17], where we also provide **supporting materials** such as a language manual for an easy application of

[4] **A**ffordable **S**afe and **S**ec**U**re **M**obility **E**volution. http://assume-project.eu/

the language. Within this paper, we will shortly introduce the three general pattern types of SSPL that are used to express functional system behavior:

Global Invariant Patterns allow the definition of conditions that need to hold without any constraints. They have no restricted scope and need to be fulfilled at all points in time. An example for this type of pattern would be

> *supply_voltage* **is always less than or equal** *14V*.

Trigger-Reaction Patterns specify system behavior that stands in some trigger-reaction relation to each other. That is why this pattern type is the most important one when it comes to the specification of system behavior. In SSPL, the reaction must occur at some point in time *after* the trigger is fulfilled, even if the reaction time is instantiated with e.g. 0ms. If a simultaneous reaction is desired, please refer to the *Simultaneity Patterns*.
The Trigger-Reaction Patterns always feature the basic structure '**Whenever** <*trigger*> **then in response** <*reaction*> **within** <*time*>', where the *trigger* and *reaction* parts can be replaced by various expressions. A possible pattern instance would be

> **Whenever** *sys_temp* **increases above** *120°C* **while** *temp_warning* **is** *'OFF'* **then in response** *temp_warning* **changes to** *'ON'* **within** *50ms* **and then** *sys_state* **changes to** *'CRITICAL'* **within** *30ms*.

Simultaneity Patterns describe the dependent fulfillment of two or more conditions at the same points in time. Equivalent to the logical implication, these patterns allow expressions like

> **While** *voltage* **is less than** *3V***,** *start_up* **does not occur.**

Each general pattern type features a variety of possible instantiations to support a broad band of different system behaviors. Furthermore, SSPL is designed to use an existing **system architecture as ontology** to derive suitable keyword replacements such as interface names. Combining that with a **scope restriction** on the system component to be described results in an overall increased quality of specifications.

4.2 Introducing Macros

Sometimes it is unavoidable to use complex expressions within a pattern language, where a natural language expression would be much shorter or easier to read and understand. That is

why we introduce the concept of typed *macros*, which extends the expressiveness of our pattern language towards a DSL while maintaining readability. Other than the approach of the already existing template languages [Gr08, Au15], we oblige the user to define a meaning for each natural language phrase by specifying a corresponding pattern language expression. Macros are not merely text replacement, but are typed according to a class hierarchy. For instance, *event* is a built-in type of our language, and the domain engineer can derive a subclass *failure event* from it. This way we ensure that even with natural language elements, all built expressions within our pattern language have unambiguous semantics. Macros can only replace a non-terminal from the BNF, as the semantics is only guaranteed to be specified on that level. Terminals can have different meanings due to their context and can thereby not be used as a macro definition.
To demonstrate the possible advantages of macros, we first translate the following example without using them:

> **NL:** Whenever any system critical error occurs the system must enter the safe mode within 30ms.
> **SSPL: Whenever any of the following events occur:**
> - *sys_err1* **occurs**
> - *sys_err2* **occurs**
> - *sys_temp* **increases above** *120°C*
>
> **then in response** *sys_mode* **changes to** '*SAFE*' **within** *30ms*.

We now want to simplify the pattern expression by replacing the event list with a macro. To do this, we need to look into the syntax definition of the pattern language and search for the corresponding non-terminal expression

> <any_event_occurs>: **any of the following events occurs:** <event_list>

We found out that the corresponding non-terminal is <any_event_occurs>, which will be the type of our new macro we want to define next. To be able to identify a macro more easily when applied in a pattern expression we chose to underline it

> 'any system critical error occurs' is defined as:
> **any of the following events occur:**
> - *sys_err1* **occurs**
> - *sys_err2* **occurs**
> - *sys_temp* **increases above** *120°C*

After defining the macro, we can now use it within our pattern language in every place, where the non-terminal <any_event_occurs> can be inserted. The resulting expression using the macro now looks very similar to the original natural language expression

> **NL:** Whenever any system critical error occurs the system must enter the safe mode within 30ms.
> **SSPL: Whenever** any system critical error occurs **then in response** *sys_mode* **changes to** '*SAFE*' **within** *30ms*.

To be even more similar to the original expression, we can define a macro of the type <var_change> for the change to safe mode. The resulting template expression looks nearly identical to the original one

> 'the system enters safe mode' is defined as:
> *sys_mode* **changes to** '*SAFE*'
>
> **NL:** Whenever any system critical error occurs the system must enter the safe mode within 30ms.
> **SSPL: Whenever** any system critical error occurs **then in response** the system enters safe mode **within** *30ms*.

When using macros it must be considered that there exist some drawbacks in comparison to pattern expressions without them. Macros introduce a layer of abstraction to the textual representation of the expression, as actual interface names and values can be masked behind a natural language phrase. That given, the reader can not directly identify those key elements and must look for the definition of the macros. There is also the possibility that a bad macro name is chosen, which could lead the reader to a false interpretation. To be completely sure on the meaning of an expression that uses macros, studying their definitions is essential.

5 Method and Tool Integration

Only providing the raw templates without any further guidance may result in user despair. The large number of possible statements or the variety of expressions can easily confuse the user, making the work harder instead of bringing more ease to it. In addition, the user might even be unsure at which step during the development process the templates should be used. At some process steps, it might not be useful or even impossible to apply template expressions.
In the following section, we provide solutions to all of the problems above. We introduce a development process that combines the component- and contract-based design approach with our previously proposed pattern language. Furthermore, we present our prototypical Tool-Framework SAVONA, which implements the described development process and allows an easy application of our pattern language.

5.1 System Development Process Using Contract-Based Design

To exploit our template language for integrated architecture refinement, we suggest following the contract-based top-down process first presented in Kaiser at al. [Ka15]. The chosen

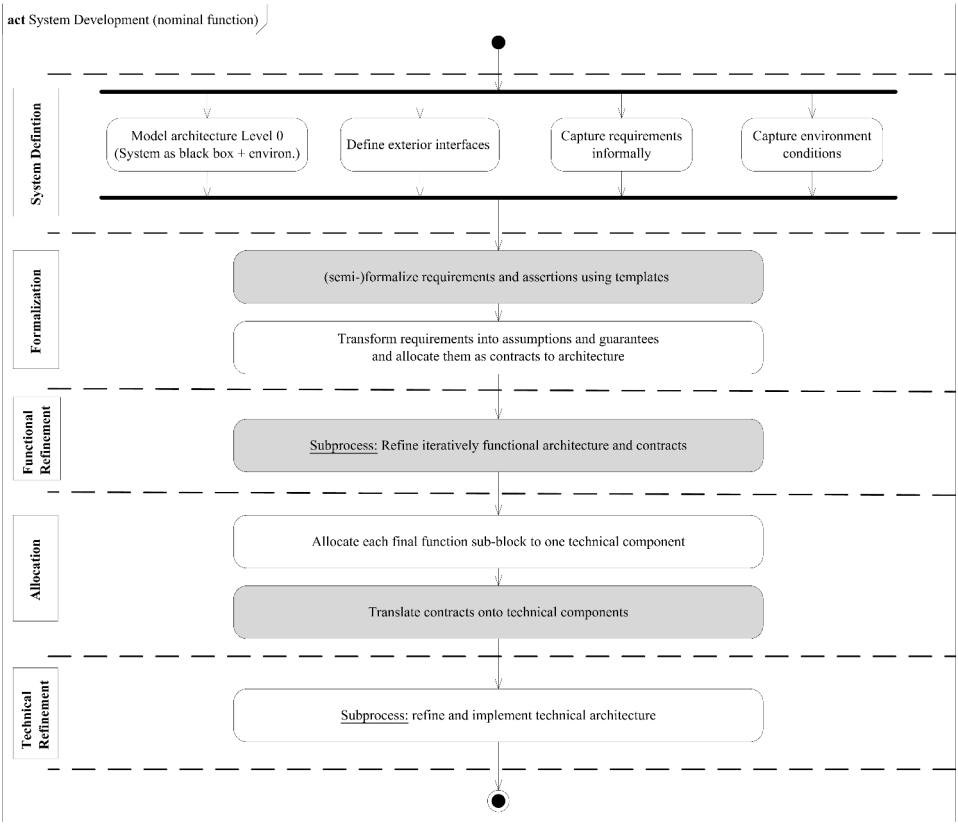

Fig. 2: Activity diagram showing the phases of the suggested contract-based development process of [Ka15]. Gray activities include the application of SSPL.

development process can be applied to the standard V-Model process and consists of following phases: *System Definition*, *Formalization*, *Functional Refinement*, *Allocation* and *Technical Refinement* (see Fig. 2). We describe each of them with regards to the interplay with our template language.

System Definition Phase At the beginning of the development process, the system engineer creates a static model of the system architecture. For that, the top-level interfaces of the system need to be defined, as they are essential for the interaction with the systems environment. The system to be developed is first modeled as a black box, because at the current development phase there is no information given on the inner workings of

the system. Aside, the top-level system requirements need to be specified. It is advised that they are captured in natural language at this development phase, as it is much faster and sufficient for the first attempt on gathering relevant information. In addition to the requirements on the system, environmental conditions do also need to be recorded. They are later used to form assumptions on inputs the system has to work with and to formulate guarantees the system has to assure in order to fulfill the environments' assumptions.

Formalization Phase Next, the requirements and assumptions are formalized using our SSPL language, which should be supported by a convenient tool, offering natural language typing with syntax highlighting and/or a template-wizard with pick lists. We use the existing *system architecture as ontology* to get information for filling out the templates, e.g. available interfaces serve as variable names to replace the corresponding placeholders and available components in the model are potential subjects to requirements. SSPL formulates assertions, which can be used in the role of an assumption or a guarantee and can be allocated to any component within the architecture (at first, the system, which is the top-level component). It is assured by the tool that only behavioral aspects that are visible on the current scope can be mentioned in the assertion, i.e. talking about internal variables or interfaces is not allowed, as well as talking about foreign components. The possibility to use macros and definitions keeps the assertions compact and readable.

Functional Refinement Phase Until now the system has only been modeled and specified at its top level. In the functional refinement phase, the system will no longer be considered as a black box, but is refined and modeled by using sub-components within the functional architecture.
This step involves design decisions by the engineer, as he or she comes up with suggestions on components, which realize the functionality of the current system level, which are also described on black-box level, addressing only the visible behavior at the ports. Each new sub-component is initially described by a natural language description and a feature list. The external interface (the ports) is defined and connected to other components via signals. In order to budget reaction times or value ranges more easily it is suggested first to assign assertion to the signals. This involves some arbitrary assignment of sub-functions to components and some arbitrary budgeting of reaction times, accuracy, ASIL etc., which is all part of the architect's design decisions and should be guided by experience what is feasible for the later technical implementation of the components. After this assignment and budgeting, new assumptions and guarantees are formed as template expressions based on the signal and super-block assertions but tailored to the scope of the component they are assigned to. This means that only interface names are available for usage in the template expressions if there exists the corresponding interface on that exact component. The formalized assumptions and guarantees are bundled as one or more contracts and assigned to the corresponding components. The contracts are then validated and verified against the architecture and the other assigned contracts. The validation includes a *compatibility check*, which verifies that only ports with compatibly types are connected, and a *consistency check*, which verifies that each constraint in a contract is satisfiable. The most important

verification step compares contracts on lower architecture level against contracts on the next higher level and checks whether the sub-components contracts allow the satisfaction the super-components contracts. This is called *refinement check* and must in most cases be performed manually today, by a review that is guided by the tool on detailed level. The more patterns of our language can be underpinned with formal semantics, the more of this type of verification can be performed formally. Detailed information about the formal refinement check of contracts can be found e.g. in [CT12]. The refinement is repeated iteratively.

Allocation Phase When the architecture has reached a level of details where actual technical components can take place of the lowest-level subcomponents, the allocation phase starts. Each black box component is replaced by one technical component with matching interfaces. The assigned contracts of the black box component become the requirements for the technical implementation and the verification obligations for the technical components. For each replacement, a final refinement check must be done to verify that the system contracts are still valid.

Technical Refinement Phase After specifying the system and modeling the static architecture, it now comes to the creation of dynamic models. Each technical component of the static architecture needs to be refined with a dynamic model (e.g. state diagrams) which represents the behavior based on its contracts' specification. The dynamic model is refined iteratively until a sufficient model depth is reached and verification is performed with existing means (e.g. testing, model checking, simulation) to show for each technical component that it fulfills its guarantees, provided that the assumptions hold.

As shown in [Ka15], the advantage of the contract-based model-integrated approach is that it works not only in a top-down manner, as suggested by the V- or Waterfall-Model, but also bottom-up. As long as pre-existing components are annotated with assumptions and guarantees, they can be stored in a library and reused later in a new context, and after re-executing the incremental verification, it becomes clear whether the resulting system is correct and safe, or the component has to be modified or replaced by another component due to detected inconsistency. As this plug-and-play approach works quite fast and contract violations are visible immediately, it supports agile development approaches to safety-critical automotive systems in an ideal way. In this case, the claim is usually not to write a complete specification, but only to fix the properties in contracts that absolutely must be guaranteed (e.g. to fulfill the safety case) and leave the rest as flexible design decisions to the development team.

5.2 Tool Support

Combining our methodological and template language approaches resulted in the prototypical tool-framework SAVONA. Based on Papyrus[5], it features the **creation of system models**

[5] https://eclipse.org/papyrus/

as Internal Block Diagram (IBD), which is provided by SysML. Due to its appearance, it fits perfectly into the component-based design paradigm and fulfills the property to model a static system architecture. In addition, Internal Block Diagrams can be used either in the system architecture, component architecture or in the hard- and software architecture by using the same model elements. As SysML is widely used as a modeling language for system engineers, the users of our tool-framework do not need to get used to any new modeling language. Additionally, various **verification mechanisms** have been implemented to ensure the validity of the modeled architecture, such as the detection on inconsistent port assignments, detection of invalid connectors, and the detection of cycles within the system architecture. We thought about different ways to **support the user at writing template**

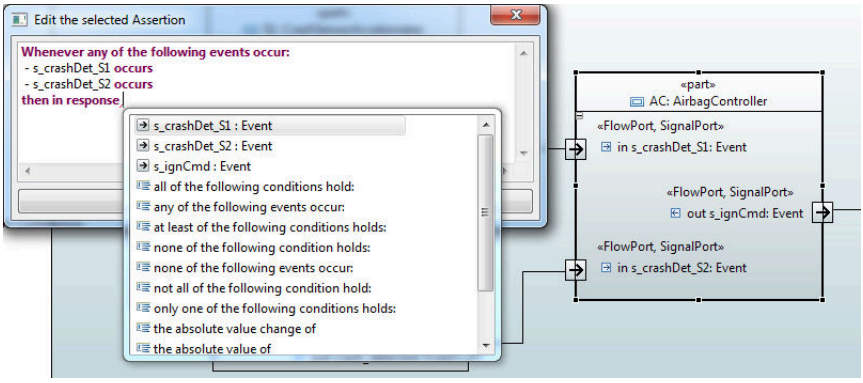

Fig. 3: *Assertion Editor*: Using the IBD's static system architecture model as ontology for SSPL pattern instantiation in SAVONA.

assertions and ended up providing two ways, which allow the user to specify assertions by using our patterns more easily. The first option the user has is to use an *Assertion Wizard*, which guides him or her through a preselected set of available pattern constructs together with examples. If the user has decided on a pattern, he or she just needs to adjust minor details such as variable names or conditional relations until the assertion is completed. The other option is to directly type assertions in a text editor called *Assertion Editor* (see Fig. 3), which features automatic syntax checks content assistance and auto-completion. A more detailed description of the SAVONA tool and its features can be found in [Gr17].

6 First experience with case study and industry projects

After the initial benchmark of our template approach by translating an automotive light system specification provided by Daimler, we are currently applying the method and tooling on two case studies. Both are provided by us within the AMASS[6] project. One of them is a platoon of model cars with a 1:8 scale, where we want to cope with autonomous driving by

[6] Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems, https://www.amass-ecsel.eu/

developing a cooperative & adaptive cruise control (CACC).
The other case study we are contributing is a standalone direct current (DC-)motor drive system, which is used in the model cars. This system is so simple that it allows an easy understanding and verification, while still exhibiting all relevant properties of a typical embedded system, in particular the combination of discrete-state logic with event typed input signals and continuous-value dynamics with continuous input and output signals. By applying our proposed model- and contract-based development approach we are are facing various system properties to design and specify. To give an example, we present the following natural language requirement that describes a part of the DC-Drives control unit:

> 'If the measured rotational speed *Spd_Act_Meas* is less than 1 rpm for more than 20ms and the rotational speed target *Adj_Spd_Tgt* is equal 0 rpm then the voltage output *V_Mot* to the motor shall be reduced to a range of [0.1V,1V] within 10ms and stay within that range for at most 15ms. After that, the voltage output will be set to 0V and remain so until the rotational speed target greater than 1 rpm for a duration of at least 50ms.'

Fig. 4 shows a possible signal trace that fulfills the given requirement. An appropriate translation with SSPL results in the following:

Whenever *Spd_Act_Meas* **is less than** 1rpm **for more than** 20ms
and (*Adj_Spd_Tgt* **is** 0rpm **)**
then in response
V_Mot **is always in the range from** 0.1V **to** 1V **for at most** 15ms
starting after at most 10ms
and then (*V_Mot* **is always** 0V **starting without any delay**
until (*Adj_Spd_Tgt* **is greater than** 1rpm **for at least** 40ms **)).**

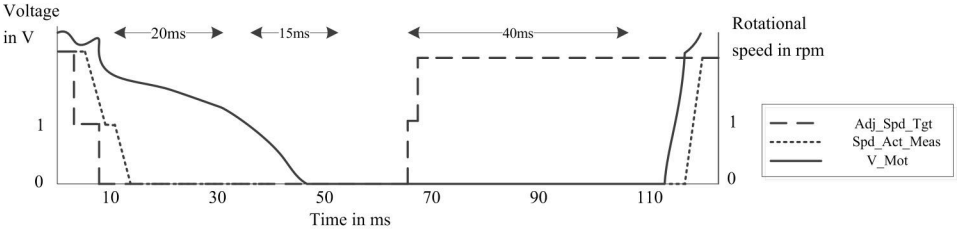

Fig. 4: Possible signal trace fulfilling the given requirement from section 6.
In the first 10ms, the target speed decreases to 0rpm and thereby stops the rotation of the motor as voltage on V_Mot is decreased. In the interval [10ms:30ms], the measured speed target speed are 0, resulting in the fulfillment of the trigger.
With a couple milliseconds delay, the motor voltage enters the given range and reaches 0V with a duration less than 15ms. After that, a new target speed is set and holds for 40ms, resulting in an increasing output voltage and measured rotation speed.

7 Conclusions and outlook

The presented approach and prototypical tool addresses a large part of the requirements listed in the introduction section. Furthermore, first experiences with applying the approach and tool onto a small DC motor drive system, as well as a more complex case study of an autonomous and networked model car, including its CACC / Platooning function have been made. Although even the latter case study is still much simpler than actual vehicle systems, we could gain a good impression about the applicability to real-world automotive systems. However, even these small case studies show that there is still a long way to go towards an industry-mature specification and modeling approach. One enhancement with high priority is to implement a formal check of the refinement for as many assertion patterns as possible, in order to unburden the developer with the manual review steps necessary today. Verification possibilities on lowest level should also be kept in mind, e.g. by enabling the automated generation of observers from the pattern expressions, which check the compliance of actual (model-in-the-loop) simulation runs or (hardware-in-the-loop) test runs. As the specification languages on system, component, hardware, and software level are quite different, it is desirable to extend the pattern language by a set of predefined macros towards a variety of domain specific languages to improve user acceptance. Regarding the extension of semantics, we are working especially on further patterns for continuous signal properties. Examples are properties like stability or bounded output signal range, or the settling time of a controller. A potential extension we are investigating is to provide patterns for frequency domain properties. Another extension of our approach would be directed towards structured data types at interfaces plus specification patterns for set, quantifier, and ordering information, in order to deal with object lists etc. For safety and reliability/availability assertions, but also in order to specify the nominal properties of environment-perceiving sensors, probabilistic patterns should be provided. Potential verification approaches for these assertions could be probabilistic model checking, but, more promisingly, Monte-Carlo simulation of the underlying behavioral models. To bridge the different levels of abstraction, a tool will have to offer different views with the option to hide details or aspects that are not of interest on a higher level of abstraction. This could on long term be complemented by a sort of 'approximate refinement' that relaxes the verification step from one level of abstraction toward the next lower level by allowing that the refined system not fully complies with the specification on higher level, but only 'well enough'.
Regarding the transition from ES to CPS and SoS, the technique should ideally be extensible towards runtime certification mechanisms, i.e. it should be possible to specify different sets of assumptions and guarantees as meta-information at runtime, so that partial systems willing to cooperate can check in which constellation assumptions and guarantees match, so that a template safety case prepared at development time is fulfilled at runtime, meaning that safe operation is assured.

References

[Au15] Autili, Marco; Grunske, Lars; Lumpe, Markus; Tang, Antony: Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar. IEEE Transactions on Software Engineering, 41(7):1-1, 2015.

[Bo15] Boyer, Benoît; Quilbeuf, Jean; Etzien, Christoph; Marazza, Marco; Senni, Valerio; Stramandinoli, Francesca; Peikenkamp, Thomas: GCSL syntax, semantics and meta-model. Technical report, DANSE Research Project, 2015. DANSE Deliverable 6.3.3.

[CE10] CESAR: Definition and exemplification of RSL and RMM. Deliverable D_SP2_R2.1_M1, Costefficient methods and processes for safety relevant embedded systems. Technical report, CESAR, April 2010. Zugriff am 29.08.2016.

[Ci12] Cimatti, Alessandro; Roveri, Marco; Susi, Angelo; Tonetta, Stefano: Validation of requirements for hybrid systems: A formal approach. ACM Transactions on Software Engineering and Methodology (TOSEM), 21(4):22, 2012.

[CT12] Cimatti, A.; Tonetta, S.: A Property-Based Proof System for Contract-Based Design. In: Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on. pp. 21–28, Sept 2012.

[DSS12] Daramola, Olawande; Sindre, Guttorm; Stalhane, Tor: Pattern-based security requirements specification using ontologies and boilerplates. In: Requirements Patterns (RePa), 2012 IEEE Second International Workshop on. IEEE, pp. 54–59, 2012.

[Gr08] Grunske, Lars: Specification patterns for probabilistic quality properties. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. pp. 31–40, May 2008.

[Gr17] Grabowski, Markus: Why Templates on System Behavior Are Not Used in Practice Yet: A Proposal for Enhancements, Application and Formalization. Master's thesis, Technische Universität Berlin, 2017.

[HJD04] Hull, Elizabeth; Jackson, Ken; Dick, Jeremy: Requirements Engineering. Springer, 2004.

[IKD09] Ibrahim, Noraini; Kadir, Wan MN Wan; Deris, Safaai: Propagating requirement change into software high level designs towards resilient software evolution. In: Software Engineering Conference, 2009. APSEC'09. Asia-Pacific. IEEE, pp. 347–354, 2009.

[Ka15] Kaiser, Bernhard; Weber, Raphael; Oertel, Markus; Böde, Eckard; Monajemi Nejad, Behrang; Zander, Justyna: Contract-Based Design of Embedded Systems Integrating Nominal Behavior and Safety. Complex Systems Informatics and Modeling Quarterly (CSIMQ), 2015 (4):66–91, Oct 2015.

[Ma09] Mavin, Alistair; Wilkinson, Philip; Harwood, Adrian; Novak, Mark: Easy approach to requirements syntax (EARS). In: Requirements Engineering Conference, 2009. RE'09. 17th IEEE International. IEEE, pp. 317–322, 2009.

[Me92] Meyer, Bertrand: Applying'design by contract'. Computer, 25(10):40–51, 1992.