On Design-time Modelling and Verification of Safety-critical Component-based Systems

Nermin Kajtazovic, Christopher Preschern, Andrea Höller, and Christian Kreiner

Institute for Technical Informatics, Graz University of Technology,
Inffeldgasse 16, 8010 Graz, Austria

E-mail: {*nermin.kajtazovic, christopher.preschern, andrea.hoeller, christian.kreiner*}*@tugraz.at*

Abstract

Component-based Software Engineering (CBSE) is currently a key paradigm used for developing safety-critical systems. It provides a fundamental means to master systems complexity, by allowing to design systems parts (i.e., components) for reuse and by allowing to develop those parts independently. One of the main challenges of introducing CBSE in this area is to ensure the integrity of the overall system after building it from individual components, since safety-critical systems require a rigorous development and qualification process to be released for the operation. Although the topic of compositional modelling and verification in the context of component-based systems has been studied intensively in the last decade, there is currently still a lack of tools and methods that can be applied practically and that consider major related systems quality attributes such as usability and scalability.

In this paper, we present a novel approach for design-time modelling and verification of safety-critical systems, based on data semantics of components. We describe the composition, i.e., the systems design, and the underlying properties of components as a Constraint Satisfaction Problem (CSP) and perform the verification by solving that problem. We show that CSP can be successfully applied for the verification of compositions for many types of properties. In our experimental setup we also show how the proposed verification scales with regard to the complexity of different system configurations.

Keywords: component-based systems; safety-critical systems, compositional verification, constraint programming.

1. Introduction

Safety-critical systems are controlling the technical processes in which certain failures may lead to events causing catastrophic consequences for humans and the operating environment. Automotive, railway, and avionics are exemplary domains here, just to name few. In order to make these systems acceptably safe, their hardware/software engineering has to be rigorous and quality-assured.

Currently, rapid and continuous increase of systems complexity represents one of the major challenges when engineering safety-critical systems. The avionics domain for instance has seen an exponential growth of software-implemented functions in the last two decades ([6]), and a similar development has also occurred in other domains with a focus on mass production, such as automotive or biomedical engineering ([16]). In response, many domains have shifted towards using component-based paradigm ([24], [10]). The standards such as the automotive AUTOSAR and IEC 61131/61499 for industrial automation are examples of widely used component systems. This paradigm shift enabled the improvement in reuse and reduction of costs in development cycles. In contrast to traditional paradigms such as the procedural and the object-oriented programming, in CBSE more attention is given on systems engineering for parts of the system rather than considering the system as a whole, i.e., on developing components. This opens many opportunities for developers and maintainers, such as more precise control and traceability over parts of the system, and possibility on their systematic reuse, which goes beyond the plain add-hoc reuse of code, objects and libraries. In some fields, the modularity of the system structure is utilized to distribute the development across different roles, in order to perform many engineering tasks in parallel. For instance, the automotive manufacturers are supplied by individually developed middleware and devices which can run their applications.

On the other side, the new paradigm also introduced some new issues. One of the major challenges when applying CBSE is to ensure the integrity of the system after building it from reusable parts (components). The source of the problem is that components are often developed in the isolation, and the context in which they shall function is usually not considered in detail. In response, it is very difficult to localize potential faults when components are wired to form a composition – an integrated system ([12]), even when using quality-assured components. The focus of the current research with regard to this problem is to enrich components with properties that characterize their correct behavior for particular context, and in this way to provide a basis for the design-time analysis or verification[*]of compositions ([8]).

This verification is also the subject of consideration in some current safety standards. For instance, the ISO 26262 standard defines the concept Safety Element out of Context (SEooC), which describes a hardware/software component with necessary information for reuse and integration into an existing system. Similarly, the Reusable Software Components concept has been developed for systems that have to follow the DO-178B standard for avionic software. These concepts both share the same kind of strategy for compositional verification: contract-based design. Each component expresses the assumptions under which it can guarantee to behave correctly. However, the definition of the specific contracts, component properties and validity criteria for the composition is left to the domain experts.

From the viewpoint of the concrete and automated approaches for compositional verification and reasoning, many investigations have focused on behavioural integrity, i.e., they model the behaviour of the components and verify whether the composed behaviours are correctly synchronized ([2]), ([4]). On the other side, compositions are often made based on data semantics shared between components ([5]). Here, the correct behaviour is characterized by describing valid data profiles on component interfaces. In both cases, many properties can be required to describe a single component and therefore scalability of the verification method is crucial here.

In this paper, we present a novel approach for verification of compositions based on the data semantics shared between components.[†] We transform the modelled composition along with properties into a Constraint Satisfaction Problem (CSP), and perform the verification by solving that problem. To realize this, we provide the following contributions:

- We define a component-based system that allows modelling properties within a complete system hierarchy.
- We define a structural representation of our modelled component-based system as a CSP, which provides us a basis to verify the preservation of properties.
- We realize the process that conducts the transformation of the modelled component-based system into a CSP and its verification automatically.

The CSP is a way to define the decision and optimization problems in the context of Constraint Programming paradigm (CP) ([3]). Using this paradigm for our component-based system, many types of properties can be supported. Also, various parameters that influence the scalability of the verification

[*]In the remainder of this paper, we use the term *verification* for the static, design-time verification (cf. static analysis ([25])).
[†]This article is an extended version of our previous work ([14]).

can be controlled (used policy to search for solutions for example). In the end of paper, we discuss the feasibility of the approach with regard to its performance.

The remainder of this paper is organized as follows: Section 2 describes the problem statement more in detail and gives some important requirements with regard to modelling a system. In Section 3 and 4, the proposed approach to systems modelling and verification is described. Section 5 describes the experimental results. A brief overview of relevant related work is given in Section 6. Finally, the concluding remarks are given in Section 7.

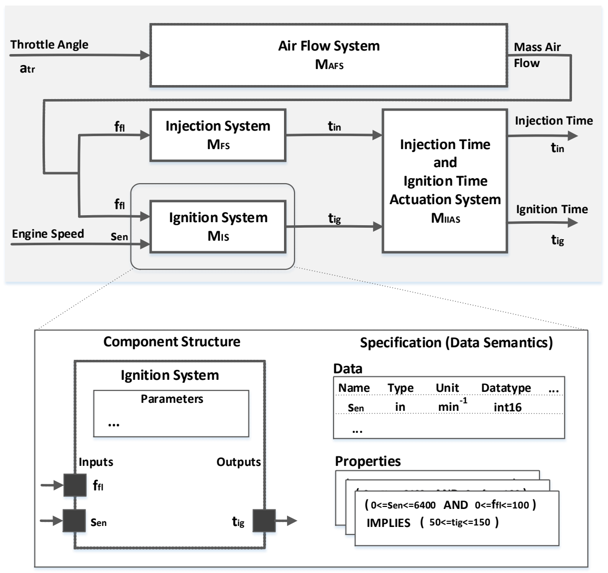

Fig. 1. Motivating example: a component-based system of automotive engine control function, adopted from ([11]) (top), and detailed view of the component Ignition System (structure and specification, bottom).

2. Problem Statement

Properties are an important means to characterize functional and extra-functional aspects of components. Safety, timing and resource budgets are examples here, just to name few ([21]). Recently, they get more and more attention in the safety community, since efficient (an practical) methods for reuse and system composition are crucial in order to reduce costs in development cycles and costs for certification of today's safety-critical systems (i.e., their extensive qualification process). In this section, we give an insight into the main challenges when using properties to verify compositions, and based on these challenges, we outline the main objectives that we handle in this paper.

2.1. Motivating Example

In our work, we address properties that in general describe data semantics. To clarify this, let us consider now the example from Figure 1. The system in this figure shows the composition of four components that form the automotive engine control application on a higher abstraction level. The basic function of this application is to decide when to activate the tasks of the fuel injection and ignition ([11]). To do this, the application takes the sensed values of the air flow volume, current speed and some parameters computed from the driver's pedal position. In a typical automotive development process,[‡] the system structure from figure is made based on stepwise decomposition of top-level requirements, having several intermediate steps such as the functional and technical system architecture with several levels in the hierarchy ([20]). Let us assume now that involved components are already developed, eventually for the complete car product line, and are stored in some repository. Let us further assume that we have a top-level requirement with regard to the engine timing for particular car type, which states the following:

The minimal allowed time delay between the task of the fuel injection and ignition shall be greater than 40 ms.

The main contributors to this requirement are software components M_{AFS}, M_{FS}, M_{IS}, M_{IIAS}, and their execution platform (e.g., concrete mapping of components on real-time tasks, task configurations, and other). In order to satisfy this timing property, the developer has to analyze the specification for each component in order to find the influence of the component behaviour on that property. The example of such a specification is given in Figure 1, bot-

[‡]Note that we do not limit our approach to automotive domain.

tom. Here, the context for the component Ignition System is defined in terms of the syntax and semantics related to component inputs, outputs and parameters. With the properties shown below, the concrete behavior can be roughly described – in this example, for certain intervals of inputs, the component can guarantee that the output t_{ig} lies within the interval $[50,150]$ (note that pseudo syntax is used here). When building compositions based on such properties, the developer has to consider their influence on the remaining, dependent components – in this case, it has to be decided whether the M_{IIAS} component can accept such values of the t_{ig} and what should components M_{FS} and M_{AFS} provide so that higher delay than $40ms$ between t_{ig} and t_{in} can be achieved. This can be very tedious and error prone task when doing it manually, because of the following reasons:

- Many components may be required to build a complete system, depending on their granularity. For example, current automotive systems comprise several hundreds of components, and many of them may depend on each other ([16]).
- Some components that directly influence the safety-critical process are usually certified, i.e., developed according to rigorous rules from safety standards. Because of costs for such a certification, the practice is to develop components for different context and to certify them just once (e.g., to support different engine types in our example). In response, many properties have to be defined for a single component to capture all context information.

The main problem here is how to define and to inter-relate all properties thorough the complete system hierarchy in a way that the preservation of properties of all components can be verified automatically? Another problem is how to complete with such a verification in a "reasonable time"?

2.2. *Modelling and Verification Aspects*

To narrow the problem statement above, very important prerequisite to structure properties within a system hierarchy consistently is to define basic relations among them. For example, properties of the component M_{IS} are related with properties of the component M_{IIAS}, because of direct connections between their output and input variables. On the other hand, properties of all four components influence the semantics of the mentioned top-level property. We summarize different types of these relations as following:

- *Composition*: hierarchical building of composed properties based on their contained properties (e.g., the top-level timing property is composed of properties contained in components M_{AFS}, M_{IS}, M_{FS} and M_{IIAS}). We discuss this later in more detail.
- *Refinement/abstraction*: properties characterize the component behaviour at certain abstraction level. With refined properties, more specialized behaviours can be described. For example, the property in Figure 1 may include some additional parameters to define conditions for the t_{ig} more precisely.
- *Alternatives*: properties may have alternative representations for different context (e.g., the Injection System component M_{IS} can provide different properties for different engine types).

These relations have to be supported when modelling a component-based system and they have to be considered when such a system has to be verified.

3. System Modelling, Verification and Deployment: An Overview

In this section, we summarize the workflow that integrates the proposed approach for systems modelling and verification. We use this workflow to verify the consistency of systems design, i.e., when the system is initially developed by assembling components, or when changes on that system have to be performed – such as component replacements or changes of the internal systems state, represented in terms of component or systems parameters.

In our previous work ([13]), we described a method on how to change safety-critical systems, with the aim to repair that system in the operation and maintenance phase by replacing malfunctioning software

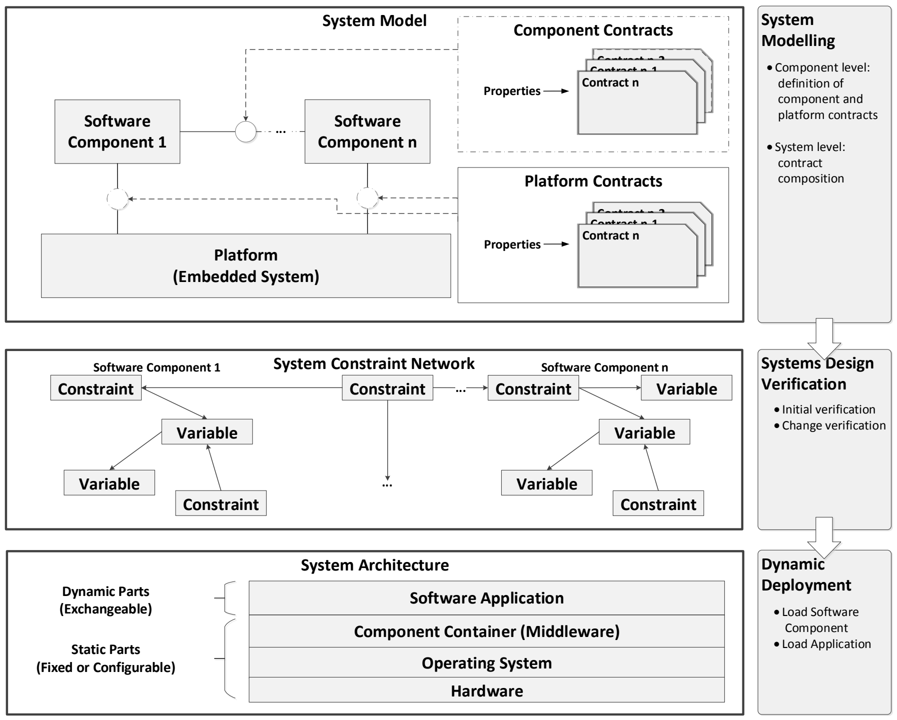

Fig. 2: Application of the proposed modelling and verification – workflow to verify an impact of changes on system integrity ([13]): system modelling using contracts (top), system design verification (middle) and dynamic deployment of software components (bottom)

components or by changing systems configuration at reduced development and maintenance costs. To this end, we defined types of supported changes and properties that have to be considered in the modelling and verification. Further, to allow to change the system in the operation we introduced a runtime support to load software components into a real-time operating systems used for safety applications ([15]). The overall workflow for the modelling, verification and deployment is depicted in Figure 2. In this paper, we focus only on modelling and verification parts of the workflow.

In the first step of the workflow, a model of a system is provided. This model basically captures properties on a level of software components, i.e., (i) to express their behaviour and relationships they have to neighbouring components, and (ii) to express relationships between components and the platform (i.e., an embedded system). Properties are here structured using contracts, which are constructs very similar to system requirements – they express what components shall do (functional) or how they shall be (extra-functional or non-functional), while at same time they define a context in which components have to satisfy those requirements. A very important role of contracts in system design is that they allow for defining specific relationships, so that the information about system integrity expressed through functional and extra-functional requirements can be maintained. Based on this fundamental feature, the impact of changes can be easily estimated and also necessary measures to handle changes can be easily identified. The next step of the workflow deals with the analysis of the system modelled using contracts. Here, a complete system is translated into a so called constraint network – a collection of inter-connected variables and constraints. This network represents contracts in a problem domain using CP. In this way, we are able to analyse whether a modelled system violates any of the contracts. In the same way, we can verify whether changes within a system design eventually require to change requirements.

Finally, the last step of the workflow is an architectural support to perform changes. To this end, we have realized a dynamic linker that is customized for the use in real-time operating-systems for safety applications. The distinct feature of this linker is that its behaviour is predictable, and the mechanism itself is designed to meet software safety regulations (please refer to ([15]) for more details).

In the following, we describe the modelling and verification parts of this workflow more in detail.

4. Constraint-based Verification

To get a rough image of the proposed approach, we highlight the modelling and verification steps in Figure 3. The input to the verification is a modelled component-based system, enriched with properties, which are structured in contracts – M_{sys} in figure. This model is further transformed into a Constraint Satisfaction Problem (CSP) – CSP_{sys} in figure, which corresponds to the problem domain mentioned in the previous section (we discuss this later). The CSP model is processed by the constraint solver, i.e., a tool to solve the CSPs, in order determine the preservation of all properties in the system. As a result, we get a decision about such a preservation. In addition, we get concrete values of data (i.e., inputs, outputs and parameters), for which properties are preserved. All steps in the process are performed automatically.

In the following, we describe how we defined each model described above. We first give some basic assumptions for our system M_{sys}. Then we describe the main elements of that system, including properties. In the end, we describe its representation as a CSP.

4.1. General: Components and Compositions

In our system, we define a component M as follows:

$$M := \left\langle \Sigma^{in}, \Sigma^{out}, \Sigma^{par}, M_c \right\rangle \quad (1)$$

, where Σ^{in}, Σ^{out}, and Σ^{par} are inputs, outputs and parameters respectively (i.e., Σ-alphabets define input, output and parameter variables in terms of datatypes, values, and some additional attributes), whereby M_c is an optional set of contained components, and is defined according to relation (1). To clarify this, we distinguish between following two types of components:

- *Atomic components*: components that can not be further divided to form hierarchies, i.e., components for which $M_c = \emptyset$. They perform the concrete computation. The Ignition System for example may contain many atomic components, such as integrators, limiters, simple logical elements and other.
- *Composite components*: hierarchical components that may contain one or more atomic and composite components, i.e., $M_c \neq \emptyset$. Note that we use the term *composition* to indicate composite components, which also may represent a complete component-based system (cf. our system in Figure 1).

The component model introduced above is typical for data-flow systems such as the ones modelled in the Matlab Simulink for example. Similar models of component-based systems are used when considering properties for resource budgets ([5]).

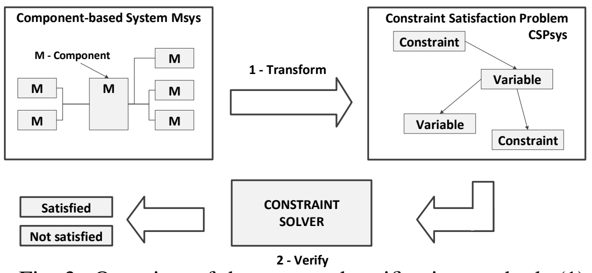

Fig. 3. Overview of the proposed verification method: (1) transformation of the component-based system M_{sys} into the CSP representation CSP_{sys}, (2) verification of the composition CSP_{sys} by solving a CSP.

4.2. Modelling Compositions Enriched with Properties

As illustrated in Figure 1, properties are defined as expressions over component variables. In order to be able to interpret these expressions during the verification, we formulate them in a SMT form[§] each expression can be represented in terms of basic symbols, such as $0, 1, ...s_{en}, ..., +, -, /, ...min$.

[§]Syntax in SMT (Satisfiability Modulo Theories) allows to define advanced expressions, e.g., on integers, reals, etc.

Using this form, various expressions can be supported for our system, including logical, arithmetic, and other. The property from Figure 1 for instance, $(0 \leqslant s_{en} \leqslant 6400) \wedge (0 \leqslant f_{fl} \leqslant 100)$, conforms to the SMT form.

In order to link properties throughout the system hierarchy with regard to three basic relations introduced in Section 2.2, we encapsulate them in assume/guarantee (A/G) contracts. According to the general contract theory in ([5]), a contract C is a tuple of assumption/guarantee pairs, i.e.:

$$C := \langle \Sigma, A, G \rangle \tag{2}$$

, where A and G are expressions over sets of variables Σ. In this way, we can split properties for each component in (a) part that has to be satisfied, i.e., *assumptions*, and (b) part that is guaranteed if assumptions hold, i.e., *guarantees*. For example, the top-level contract C_{II} for our system in Figure 1 guarantees the 40*ms* delay under assumptions that the rotational speed s_{en} and values for the throttle angle a_{tr} are within certain ranges:

$$C_{II} = \begin{cases} \text{variables} & \begin{cases} \text{inputs} & s_{en}, a_{tr} \\ \text{parameters} & - \\ \text{outputs} & t_{in}, t_{ig} \end{cases} \\ \text{types} & s_{en}, a_{tr}, t_{in}, t_{ig} \in \mathbb{N} \\ \text{assumptions} & (0 \leqslant s_{en} \leqslant 6400) \wedge (0 \leqslant a_{tr} \leqslant 100) \\ \text{guarantees} & t_{ig} - t_{in} > 40 \end{cases}$$

Based on this structure, we can link properties between dependent components in a similar way it is done when wiring components using connectors (i.e., links between their input/output variables). Figure 4 shows our example system modelled using contracts. Every component provides certain guarantees which stay in relation to assumptions of dependent components. These components in turn provide guarantees based on their own assumptions, and so forth. In this way, all properties within a system hierarchy can be linked together. In Figure 4, we have also highlighted different types of relations between contracts, required to build such a hierarchy (see Section 2.2). These are:

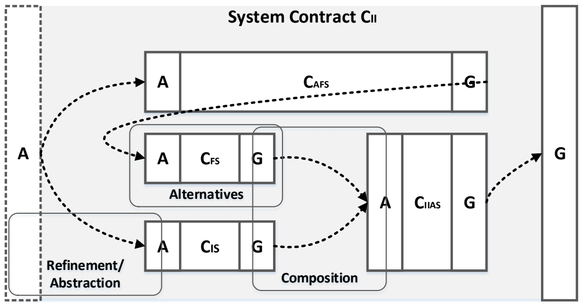

Fig. 4. The Engine Controller system represented using contracts and their basic relations (A – assumptions, G – guarantees, C – contracts).

- *Composition*: two contracts can interact when after connecting their guarantees and assumptions both contracts can function correctly (we discuss this in more detail in Section 4.3). We use the operator $\otimes$ to define a composition ([5]). An example of such relations is shown in Figure 4, where contracts C_{FS}, C_{IS}, and C_{IIAS} form a composite contract, i.e., $((C_{FS} \otimes C_{IS}) \otimes C_{IIAS})$.
- *Refinement/abstraction*: similar to refinement of properties, contracts refine other contracts in terms of refined assumptions and guarantees. We use the operator $\preceq$ for this relation. The top-level contract C_{II} has such a relation with the contained contracts, i.e., $((C_{FS} \otimes C_{IS}) \otimes C_{IIAS}) \preceq C_{II}$. Note that only the relation with the contract C_{IS} is highlighted here.
- *Alternatives*: when designing components for more than one context, each new context is described in a separated contract. Contracts that describe the same property for different context are alternatives. In example in Figure 4, any of contained contracts may have alternatives – here, we just highlighted C_{FS} to indicate that it may have alternative contracts.

Based on definitions for contracts and their relations, we can now define the top-level system/composition contract, C_{sys}, as follows:

$$C_{sys} := (\otimes_{i \in \mathbb{N}} C_i) \tag{3}$$

, i.e., a hierarchical composition of contracts C_i, where C_i represents further composition according to relation (3).

Finally, to relate contracts with components, i.e., the concrete implementations of contracts, we extend the relation (1) as follows:

$$M := \left\langle \Sigma^{in}, \Sigma^{out}, \Sigma^{par}, C_c, M_c \right\rangle \tag{4}$$

, where C_c is a set of contracts that the component M can implement. Based on this relation, any implementation of the C_{sys} contract represents a complete component-based system or a top-level composition. We identify this implementation as M_{sys} and use it later as a basis to define our CSP.

4.3. *Ensuring Correctness of Compositions*

For our component-based system defined previously, two contracts C_1 and C_2 can form a composition (i.e., can be integrated) when their connected assumptions/guarantees match in the syntax of their variables (i.e., datatypes, units, etc.), and when following holds:

$$G(C_1) \subseteq A(C_2) \tag{5}$$

In other words, the contract C_1 shall not provide values not assumed by the contract C_2. This relation is a basis in our CSP to verify the complete composition.

4.4. *Composition as a Constraint Satisfaction Problem*

Now, we describe how we define the composition M_{sys} as a CSP. We name our CSP representation of M_{sys} as CSP_{sys}, and define it as follows:

$$CSP_{sys} := \langle X_{CSP}, D_{CSP}, C_{CSP} \rangle \tag{6}$$

, where X_{CSP} is a finite set of variables, D_{CSP} their domains (datatypes, values), and C_{CSP} a set of constraints related to variables and constraints in C_{CSP}. In other words, the CSP represents a network of variables inter-connected with each other using constraints. The constraints set variables in relations using some operators, and in this way they form expressions. Various types of expressions can be used to define constraints (e.g., Boolean, SMT – depending on supported features of the solver). The solution of the CSP_{sys} is a set of values of X_{CSP} for which all constraints C_{CSP} are satisfied. The constraint solver performs the task of finding solutions.

In order to represent the composition M_{sys} in a CSP, we need to map the top-level contract structure ((sub-)contracts, variables, and A/G expressions) into the CSP constructs mentioned above. Important aspects of this representation are CSP definitions for (1) a type system, (2) A/G expressions or properties, (3) the structure of components and contracts and (4) the structure of compositions. We can now turn to these representations.

4.4.1. *Type System*

The CSP tools, i.e., constraint solvers, usually provide the support for several domains to represent various types of variables. Integers, reals, and sets are examples here, just to name few. In order to avoid type castings between modelled system M_{sys} and CSP_{sys}, we use the same domains for both M_{sys} and CSP_{sys}. Another reason is that the time needed for the constraint solver to solve the CSP strongly depends on a particular domain. For example, there is a significant difference in runtime when dealing with real numbers instead of integers. Therefore, we use integers for both system representations, i.e., M_{sys} and CSP_{sys}.

4.4.2. *A/G Expressions (Properties)*

Concerning the representation of values of variables in the CSP, limits have to be set on their intervals. The intervals are possible search space for the solver, and can have significant influence on solver's runtime. It is therefore important to limit the variables on smallest possible intervals.

In our CSP_{sys}, each variable which is used in an expression is represented by two CSP variables: one indicating the begin of the interval, another one for the end of that interval. The size of this interval

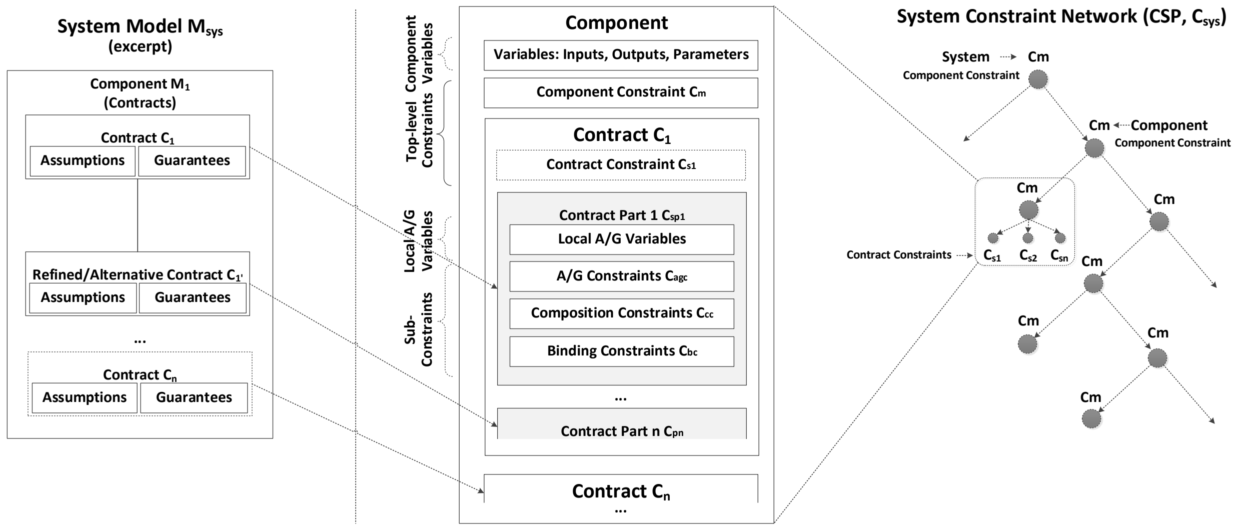
System Model M_{sys} (excerpt)
Component M_1 (Contracts)
Contract C_1
Assumptions
Guarantees
Refined/Alternative Contract $C_{1'}$
Contract C_n
Component
Variables: Inputs, Outputs, Parameters
Component Constraint C_m
Contract Constraint C_{s1}
Contract Part 1 C_{sp1}
Local A/G Variables
A/G Constraints C_{agc}
Composition Constraints C_{cc}
Binding Constraints C_{bc}
Contract Part n C_{pn}
Top-level Component Constraints Variables
Local A/G Variables
Sub-Constraints
System Constraint Network (CSP, C_{sys})
System Component Constraint
Component Component Constraint
Contract Constraints
Cm
C_{s1} C_{s2} C_{sn}

Fig. 5: Representation of a component in CSP: an exemplary component with three contracts (left) and an excerpt of the mapping of contracts to constraints and variables (right)

is determined based on intervals defined in expressions. For example, the variable s_{en} in the expression $(0 \leqslant s_{en} \leqslant 6400)$ is limited on the interval $[0, 6400]$. The reason for using two CSP variables here is that solving the CSP results with not only decision about the correctness of a composition with regard to the relation (5), but it also provides values for which the relation (5) is satisfied. In this way, we can obtain the concrete intervals (instead of just values) for all variables in all contracts (for correct compositions). This information can be useful for example when the composition M_{sys} has many alternative contracts, to observe which of them are identified as correct.

Relations or operations between variables in expressions are represented as constraints. Since both M_{sys} and CSP_{sys} use the SMT syntax for expressions, every operation is represented as a single constraint.

4.4.3. *Components*

From the perspective of the structural organization, every component is represented in a CSP as a set of variables (inputs, outputs, parameters) from the integer domain, and a set of constraints, which correspond to the contracts implemented by that component (see Figure 5).

Note that we distinguish here between variables used in components, i.e., Σ in relation (4), and variables used in contracts, i.e., Σ in relation (2). Although they are identical, we define separated variables in the CSP for each of them. This means, when a component has two contracts, we have CSP variables for (a) component variables (inputs, outputs and parameters) and (b) CSP variables (inputs, outputs and parameters) for each contract. With this separation of contracts and components, we can identify which contracts are satisfied if the verification succeeds. As mentioned, the constraint solver not only responds with a decision, but it also finds all values of X_{CSP} for which the verification succeeds. Similarly, if the verification fails, the conflicting contracts can be easily identified.

Now we describe how the contracts are defined in a CSP, how they are linked with components, and how the criteria for correctness from relation (5) is represented in a CSP.

4.4.4. *Contracts*

As shown in Figure 5, each contract is represented as a single top-level constraint C_s. This constraint is further related to a set of local A/G variables (inputs, outputs, parameters) and a set of sub-constraints. The sub-constraints represent the constraints of the refined/abstracted or alternative contracts (contract parts Cs_p in figure). Because refined/abstracted and alternative contracts do not depend on each other, we define the top-level constraint C_s as follows: $C_s := (\vee_{i \in \mathbb{N}} Cs_{pi})$. In this relation, any contract which can satisfy the relation (5) implies that the top-level contract constraint C_s is satisfied.

As illustrated in Figure 5, every contract consists of the following sub-constraints:

- A/G constraints C_{agc}: constraints related only to local A/G variables. These constraints define the assumptions and guarantees for a contract. They are defined based on A/G expressions in contracts, as described in Section 4.4.2.
- Binding constraints C_{bc}: constraints that link the local A/G variables to the global component variables so that both types of variables get the same values. In this way, we can observe which contracts were satisfied, after successful verification.
- Composition constraints C_{cc}: constraints that integrate the contracts. These constraints express the integration or composition between two contracts, as described in Section 4.3. They link two contracts according to relation (5).

All three top-level constraints have to be satisfied for a contract C_{sp}, i.e., $C_{sp} := (C_{agc} \wedge C_{bc} \wedge C_{cc})$.

Finally, the top-level constraint of a component is satisfied, if all contract constraints C_s are satisfied, i.e., $C_m := (\wedge_{i \in \mathbb{N}} C_{si})$.

4.4.5. System/Composition

The compositions have very similar structure to basic or atomic components. Because they abstract some contracts of the contained components, additional constraints are defined to link these variables. An example of such a composition is given in Figure 4, where assumptions and guarantees of the contract C_{II} are an abstraction of assumptions and guarantees of the contained contracts.

Like atomic components, the complete component-based system M_{sys} is represented in a CSP as a set of variables and constraints. Within this set of constraints, there is a single top-level constraint of the composition C_m which links the complete hierarchy of the sub-constraints and variables discussed previously (the top-level constraint C_m is shown in Figure 5 right). The CSP has a solution only if this top-level constraint is satisfied. Finally, the C_m corresponds to the top-level constraint in the constraint set C_{CSP} from the relation (6).

5. Experimental Results

In the following, we describe the results of the preliminary evaluation and we discuss the performance of our approach.

To conduct the experiment, we used Java-based Choco constraint solver ([7]). In our experiment, we defined the composition M_{sys} as a XML description, which is then used to generate the CSP in memory.

The main goal of this experiment is to show whether the proposed CSP is applicable to solve the composition problems defined with data properties, and for which system configurations. We conduct the experiment by showing how the verification responds with regard to attributes that might have an effect on runtime. These attributes include:

- Components and properties: how the verification scales with regard to number of components and properties, including also the presence of the alternative properties.
- Nature of properties: different properties may require different expressions in the CSP, including operations on fixed values, intervals, or more advanced operations such as ones used to define resource constraints (e.g., sum, min, etc.).

Figure 6 shows the system configuration used to conduct the experiments. The inputs for the verification are provided by the Environment component, which encloses the component-based system under test. All experiments were executed on Intel i7-3630QM, 4 cores, 2.40GHz.

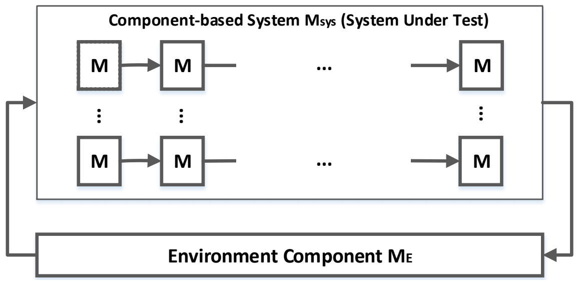

Fig. 6. System configuration used to conduct the experiments (M - component, M_E - environment component).

5.1. *Quantitative Results*

For this experiment, we performed two measurements. In the first measurement, we show the response time with regard to the number of components, properties and alternative properties, having specified assumptions and guarantees as intervals. Then, in the second measurement, we use the same configurations but with fixed values for expressions. With these two measurements, we are able to observe the limits on modeling the component-based system with regard to number of components, properties, and expressions used to describe the properties.

5.1.1. Measurements

In the first measurement, we execute several thousands of system configurations with the varying number of components and properties. The measurement has two parts. In the first part, we verify the system configurations with the varying number of components, each having varying number of properties but with constant number of assumptions or guarantees (i.e., each component variable is therefore related to only one expression). In the second part, each of the components has varying number of alternative and refined properties, so that many solutions are possible. In this case, each component variable is related to many expressions.

The expressions in the first measurement are defined in a way that always the intervals of the component variables have to be satisfied, and not the fixed values. An example for such expression is given in Section 4.2 for the contract C_{II}, which is satisfied only if the variables s_{en} and a_{tr} are in ranges $[0,6400]$ and $[0,100]$ respectively.

For the input test data, i.e., the operands of the assumption and guarantee expressions, we generate the values for each expression randomly, but with the rule that the assumptions are always satisfied. The advantage of performing the positive tests here is to get more clear statement about the runtime of the verification. In both parts of the measurement, we use the relational and logical operations on values.

In the second measurement, we execute the same system configurations as in previous measurement, but this time using the fixed values for component variables.

5.1.2. Observations

First results of the experiments are illustrated in Figure 7. On the left, an excerpt of the results for the first measurement is shown, where the properties have a constant number of assumptions and guarantees. The reason why the verification responds in short time is that each component variable has only one expression (assumption or guarantee constraint, C_{agc}), and it is then immediately instantiated to a value indicated by that expression. The runtime depends in this case therefore on the number of components and properties.

On the right in Figure 7, a scenario that is more likely to occur in practice is shown. Here, each component variable has an increasing number of expressions, and these expressions are alternatives (as mentioned in the description of the measurement). The response time of the verification strongly depends on the number of alternatives, because each of the expressions represents different interval. The solver has to adjust the component variables to adequate intervals, in order to find a solution. Furthermore, since the choice of the particular alternative may influence the choice of the intervals in other connected components, often the backtracks have to

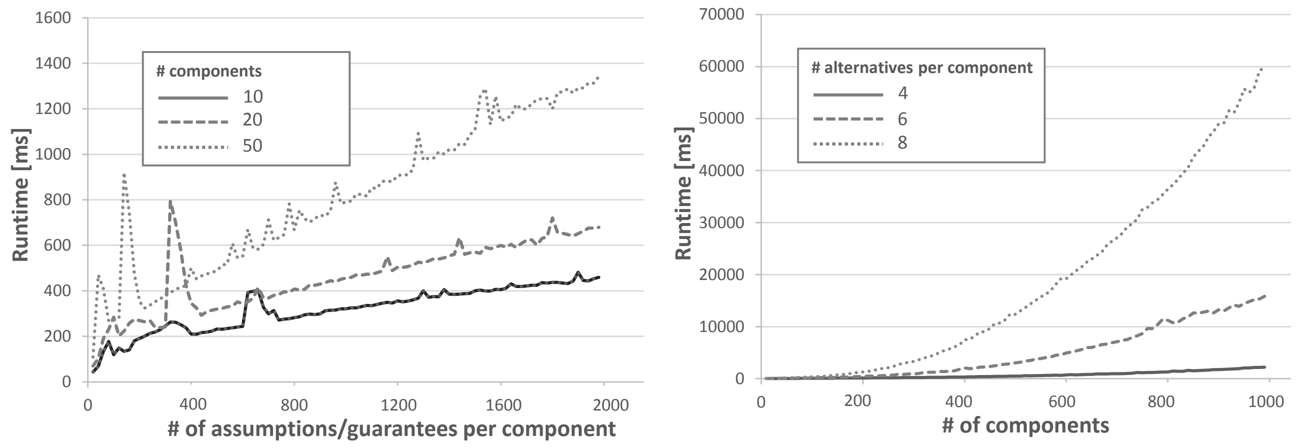

Fig. 7: Experimental results: runtime for system configuration with varying number of assumption/guarantee expressions and components (left) and varying number of components and alternative properties (right)

be done to the state where the constraints were satisfied, which is time consuming.

In the second measurement, we observed very similar results as illustrated in Figure 7 on the left. Having fixed values on component variables, no search has to be performed, but just the constraint verification. For the case where the alternatives are used, more time is required to find a solution, but this time is negligible in contrast to situation when using intervals (i.e., Figure 7, right).

In the end, we summarize our observations with Figure 8. This figure shows the region for which the verification can complete in a ”reasonable time”. We set the limit for this time on 2 minutes, just to get a first feedback about possible configurations for the system under test. To establish this region, we used the system configuration with the worst case in response time, i.e., the one having the alternative properties from the first measurement.

5.2. Qualitative Results: Discussion

Figure 8 shows the worst-case scenario, in which a component-based system is modelled having varying number of assume guarantee expressions.

The verification scales well but for configurations with only few instances of either components or properties. In nowadays automotive systems for example, there are more than 800 software components, that control various technical sub-processes in automobiles ([16]). However, it is still possible to support these configurations, since each such sub-system can be provided to verification independently, and also, not all components are massively interconnected as in Figure 6. For example, the simplified system from Figure 1 is modelled using 13 software components (is just one option to realize that system).

6. Related Work

Now we turn to a brief overview of related studies. We summarize here some relevant articles that handle compositional verification based on data semantics.

Similar problems to those described in our problem statement were identified by Sun et al. ([23]) in their work on verifying the composition of analogue circuits for analogue system design. In their approach, each analogue element (resistor, capacitor, etc.) is characterized by its performance profile and this profile is used to build the contract; that is, for certain values of the inputs the element responds with certain output values. Using contracts made from performance profiles, it was possible to eliminate many integration failures early in the system design phase. These structural compositions of analogue elements are very similar to the compositions in CBSE. However, the model of Sun et al. only considers connections between elements (horizontal relations).

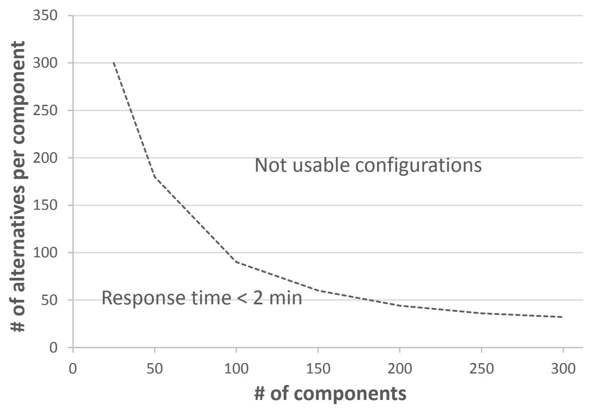

Fig. 8. Region of possible system configurations for which the verification completes within a given time.

Another article describes a runtime framework for dynamic adaptation of safety-critical systems in the automotive domain ([1]). In the event of failures or degradation of quality, the intent is to reconfigure the automotive system while it is operating. In contrast to the previous approach, the compositional verification in this case is based on a common quality type system shared among components. Two components can form a composition only when their interfaces or ports have the compatible type qualities. In this way, wrong type castings between components can be avoided. However, using a type system in our case would just verify the syntax but not the semantics of data (i.e., the concrete values).

A more advanced framework for dynamic adaptation of avionics systems was developed by Montano ([18]). The goal is to adapt the system to new, correct configurations, in case of failures. To perform this, a common quality system defines the contracts between functions and available static resources (e.g., memory consumption, CPU utilization, etc.) and in this way it restricts the possible set of correct configurations. An important aspect of this work is that it demonstrates the CSP approach to solving the composition problem. However, the quality type system only considers static resources, and does not consider contracts between functions. Ultimately, the approach is strongly focused on dynamic adaptation with human-assisted decision making.

In the field of industrial automation, the authors in ([17]) propose the static verification of compositions based on data types of the IEC 61131-3 component model (or standard). This model defines the standard data types but it also allows definition of customized data types (derived from existing ones) and combination of existing data types into complex structures. The authors identified ambiguities in the standard for user-defined data types and defined a proper compatibility criteria. Like the adaptation approach in the automotive domain ([1]), this work considers only a type system. However, the approach verifies not only compositions, but also the use of variables in IEC 61131-related languages.

In the last few years, several research projects have begun to handle the topics of compositional verification ([22]), ([9]), ([19]) by formalizing system models (component models) and languages for specification of contracts. These approaches share many concepts, especially contract-based design and formal behavioural verification of compositions. Although our model is conceptually very similar, it differs in that it considers the data semantics of property values, and it addresses a specific type of component-based systems in which data semantics can be used to express the validity criteria for compositions.

7. Conclusion

In this paper, we presented a method for modelling and verification of compositions in component-based systems. The components modelled here are enriched with properties, which describe the data semantics of components. The novelty of our verification lies in representing the composition along with modelled properties as a Constraint Satisfaction Problem (CSP), which allows us to achieve two important objectives. First, using relational, logical and more advanced operators on data, many types of properties can be supported. Second, for properties that use basic logical and arithmetic operators, the verification can scale up to several hundreds of components, each of them consisting of few tens of properties, which makes the approach promising for the use in practice.

As part of our ongoing work, we want to characterize the runtime performance based on different types of properties, since they impact the scalability at most. In addition, we also want to investigate other parameters such as solver search policy, solver engine, etc., in order to find best configuration for the verification method.

References

1. Adler, R., Schaefer, I., Trapp, M., Poetzsch, A.: Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. ACM Trans. Embed. Comput. Syst. **10**(2) (2011)
2. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes **26**(5), 109–120 (2001)
3. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)
4. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the bip framework. Software, IEEE **28**(3), 41–48 (2011)
5. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.: Contracts for Systems Design. Tech. rep., Research Report, Nr. 8147, November 2012, Inria (2012)
6. Butz, H.: Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. Dept. of Avionic Systems at Airbus Deutschland, Hamburg, Germany (-)
7. choco Team: choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO, École des Mines de Nantes (2010)
8. Clara Benac Earle: Languages for Safety-Certification Related Properties. In: WIP Session at SEAA'13
9. COMPASS: Compass - comprehensive modelling for advanced systems of systems. Homepage: http://www.compass-research.eu (2011-2014)
10. Crnkovic, I.: Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA (2002)
11. Frey, P.: Case Study: Engine Control Application. Tech. rep., Ulmer Informatik, Nr. 2010-03 (2010)
12. Gössler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comp. Prog. **55** (2005)
13. Kajtazovic, N., Preschern, C., Höller, A., Kreiner, C.: Towards assured dynamic configuration of safety-critical embedded systems. In: Computer Safety, Reliability, and Security, *LNCS*, vol. 8696, pp. 167–179. Springer International Publishing (2014)
14. Kajtazovic, N., Preschern, C., Höller, A., Kreiner, C.: Constraint-based verification of compositions in safety-critical component-based systems. In: SNPD, *Studies in Computational Intelligence*, vol. 569, pp. 113–130. Springer International Publishing (2015)
15. Kajtazovic, N., Preschern, C., Kreiner, C.: A component-based dynamic link support for safety-critical embedded systems. In: 20th IEEE International Conference and Workshops on the Engineering of Computer Based Systems, pp. 92–99 (2013)
16. Kindel, O., Friedrich, M.: Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis. dpunkt Verlag; Auflage: 1 (2009)
17. M., D.S.: Data-type checking of iec61131-3 st and il applications. In: 2012 IEEE 17th Conference on Emerging Technologies Factory Automation (ETFA), pp. 1–8 (2012)
18. Montano, G.: Dynamic reconfiguration of safety-critical systems: Automation and human involvement. PhD Thesis (2011)
19. SAFECER: Safecer - safety certification of software-intensive systems with reusable components. Homepage: http://safecer.eu (2011-2015)
20. Schäuffele, J., Zurawka, T.: Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen. V+T Verlag (2010)
21. Sentilles, S., Štěpán, P., Carlson, J., Crnković, I.: Integration of extra-functional properties in component models. In: Proceedings of the 12th International Symposium on Component-Based Software Engineering, pp. 173–190. Springer-Verlag, Berlin, Heidelberg (2009)
22. SPEEDS: Speculative and exploratory design in systems engineering - speeds. Homepage: http://www.speeds.eu.com (2006-2012)
23. Sun, X., Nuzzo, P., Wu, C.C., Sangiovanni-Vincentelli, A.: Contract-based system-level composition of analog circuits. In: Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE, pp. 605–610 (2009)
24. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley LP Co., Inc., Boston, MA, USA (2002)
25. Tran, E.: Verification/validation/certification. Carnegie Mellon University, 18-849b Dependable Embedded Systems (1999)