Towards Assured Dynamic Configuration of Safety-Critical Embedded Systems

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

Institute for Technical Informatics,
Graz University of Technology,
Infeldgasse 16, Graz, Austria
{nermin.kajtazovic,christopher.preschern,
andrea.hoeller,christian.kreiner}@tugraz.at

Abstract. Assuring systems quality is an inherent part of developing safety-critical embedded systems. Currently, continuous increase of systems complexity, in particular that of software, makes this development challenging. In response, more and more software faults are remaining unidentified at design-time so that changes and maintenance need to be performed at an increased rate. Unfortunately, today's safety-critical systems are not designed to be upgraded or maintained in a seamless way, so that the overhead of performing changes may be considerable, especially when such changes require to re-verify and re-validate the whole system.

In this paper, we present an approach to perform software changes in the operation and maintenance phase of the systems lifecycle. Changes are performed dynamically, by replacing parts of software (i.e., software components) with their functionally equal out-of-the-box instances. In order to prevent the impact of changes on systems integrity, we provide a support to model and to analyze the system. The main outcome here is that specific kind of changes can be maintained without adding any development costs.

Keywords: safety-critical embedded systems, component-based systems, dynamic configuration.

1 Introduction

Maintaining a correct function even in presence of faults is an important characteristic of safety-critical embedded systems. In order to reduce the risk of failures, and thus to avoid the potential environmental damages or harm on humans, their hardware/software development has to be rigorous and quality assured.

Currently, rapid and continuous increase of systems complexity, in particular that of software, makes the development of these systems challenging [4] [12]. In response, more and more software faults are remaining unidentified at design-time so that changes and maintenance need to be performed at an increased rate. Concrete examples of such change and maintenance demands are quite often

A. Bondavalli et al. (Eds.): SAFECOMP 2014 Workshops, LNCS 8696, pp. 167–179, 2014.

recalls of vehicles, medical devices, and other products. Some of these recalls are related to faults located in the software functions, such as the control algorithms, libraries, flaws in modification or adaptation, and other. According to recent studies related to defect analysis in recalls, those faults are getting more frequent, as more and more functions are being implemented in software [2]. Eliminating those faults in most current safety-critical systems is quite difficult, in particular because it has to be evidenced that the changed system still maintains certain level of quality – a so called safety integrity in the notation of safety standards. To provide such an evidence, many steps in the development lifecycle have to be repeated. In addition, depending on the impact of changes and regulations of the considered safety standard, new certification might be required.

In this paper, we present an approach to perform software changes in the operation and maintenance phase of the systems lifecycle. Changes are performed dynamically, by replacing parts of software (software components [5]) with their functionally equal out-of-the-box instances. Before any change can be performed, a new system configuration is analyzed against the violation of the safety integrity. Thus, only the configurations that pass this analysis step can be installed into the system dynamically. To enable such assured dynamic configurations, we have provided the following basis in our previous work: (a) a runtime mechanism that allows to load the out-of-the-box software components into a real-time operating system dynamically – the dynamic linker [10], and (b) a design-time mechanism to ensure the consistency of new system configurations [11]. This consistency mechanism performs the analysis of a changed system based on modelled properties which describe certain system attributes, such as memory and timing budgets for example[1]. In order to determine whether changes caused by replacing software components have an impact on the safety integrity, there is a need to identify which attributes may be relevant here. For this purpose, we analyze in this paper how the change management is regulated in some safety standards, and under which conditions the replacements of components are allowed.

The main outcome here is that for specific kind of changes, in which software components can be replaced, the system does not need to be turned back into the development phase. Furthermore, if the re-certification of the system is required, the original certification data can be reused, since they are not impacted by those changes. In response, replacements of software components can be maintained without any development costs.

The remainder of this paper is organized as follows: Section 2 provides a brief overview of relevant related work. Section 3 describes how changes are handled in safety standards, and which system attributes have to be considered when analyzing changes. In Section 4 the proposed approach is described, and a short discussion is given in Section 5. Finally, concluding remarks are given in Section 6.

[1] We use the notation *system attributes* to identify various functional and non-functional system aspects, such as performance requirements, constraints, etc.

2 Related Work

Now we turn to a brief overview of related studies. We summarize here some relevant articles that handle the analysis of changes in safety-critical embedded systems.

To date, much research has been done on analyzing planned changes in software architectures for safety-critical systems [1] [15] [13] . In the work by Adler et al. [1], an adaptive architecture for safety-critical automotive systems is proposed. The main goal here is to increase the systems availability by allowing software components to implement diverse behaviours, so that in the event of failures or degradation of quality, the automotive system can continue operating by switching between correct implementations. Since different implementations of components may have different quality, the authors provide a design-time analysis to prevent mixing not allowed combinations of component implementations. For this purpose, they define a quality system, with a set of fixed quality types. A more advanced framework for dynamic adaptation of avionics systems was developed by Montano [15]. 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 CP 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. Similar reconfiguration strategy is used in [13], but the consistency of the reconfiguration here is ensured by the runtime mechanisms (partitioning).

There are also some works which focus on upgrading safety-critical systems [20] [16] [19]. One of the most notable is work done in the scope of the project PINCETTE, which has as a goal to perform live upgrades of software systems that control the safety-critical processes [20]. Although the topic is beyond the scope of available validation methods in the practice, the aim is to evaluate the feasibility of formal methods to such use cases. In contrast to our data flow-oriented analysis, the focus here is on validating the interaction between upgraded behaviours. Another work [16], done in the scope of the RECOMP project, addresses also live upgrades as one of the goals to reduce the costs for certifying systems. However, only dynamic linker has been realized here, without considering the analysis of changes. Finally, the work in [19] shows how to validate changes of upgraded safety-critical system. Here, model checker is used to verify changed behaviour.

In summary, various analysis methods have been developed to validate changes. However, none of the approaches discussed here consider regulations of standards, to identify whether changes they support are allowed and, if so, to which extent.

3 Addressing Changes in Engineering of Safety-Critical Embedded Systems

Identifying system requirements affected by changes is a crucial step in the change management process. To determine which requirements and which related system attributes influence the systems safety integrity, we analyze in the following how changes are regulated in safety standards. Based on this analysis, we build a list of system attributes that we further use to construct our software architecture, and to build properties for our software components.

3.1 Change Management in Safety Standards

In general, standards for functional safety provide the guidelines on how to align the system development with the safety lifecycle in each phase. One aspect of these guidelines are activities related to maintenance and operation phase of the systems lifecycle. Changes in the operation phase are usually handled in the context of the supporting processes defined in standards, such as the maintenance, the configuration management, and the change management [18]. In the following, we describe the change management defined in the IEC 61508, which is a generic safety standard applied in the industry. We align our approach to this standard, because many guidelines it provides can also be found in other standards applied in specific industrial sectors, since they represent derivatives of the IEC 61508 (e.g., the ISO26262 standard provides similar guidelines for maintaining changes in automotive systems).

The lifecycle of the IEC 61508 standard comprises the engineering activities for software and systems scope. Changes in the operation and maintenance phase of systems are described in parts 1, 2 and 3 of the standard, in the context of the supporting processes: maintenance, configuration management, and change management. Each of these processes has defined steps, the inputs and the work products it shall produce. To ensure the safety integrity after implementing changes, the standard prescribes requirements that have to be fulfilled and a list of possible techniques and measures to apply within these processes. The requirements are mainly related to activities that need to be performed if safety integrity is affected by changes. In Table 1, we have filtered out the most relevant requirements. Basically, if safety integrity is affected by changes the standard recommends to (i) perform the hazard and risk analysis in order to identify additional faults that might be introduced by such changes and (ii) to return to the appropriate phase in the software lifecycle to implement changes. On the system level (part IEC61508-2), it is recommended to use the same development equipment and expertise (e.g., tools, previous system configuration, project artifacts, etc.), in order to just focus on changed parts only. In addition to requirements, developers have the option to choice which techniques and measures to perform, based on the level of safety integrity they want to achieve after implementing changes (bottom part of the table). Among them, the most influential measure here from the aspect of costs is a need for the verification and validation. For the highest levels of safety integrity, the standard recommends to

Table 1. IEC 61508 requirements, measures and techniques related to change management (an excerpt)

Requirements on software change management, IEC 61508-3	
7.8.2.3	An analysis shall be carried out on the impact of the proposed software modification on the functional safety of the E/E/PE safety-related system: a) to determine whether or not a hazard and risk analysis is required; b) to determine which software safety lifecycle phases will need to be repeated.
7.8.2.5	All modifications which have an impact on the functional safety of the E/E/PE safety-related system shall initiate a return to an appropriate phase of the software safety lifecycle. All subsequent phases shall then be carried out in accordance with the procedures specified for the specific phases in accordance with the requirements in this standard. Safety planning (see Clause 6) shall detail all subsequent activities.
Requirements on system change management, IEC 61508-2	
7.8.2.3	Modifications shall be performed with at least the same level of expertise, automated tools (see 7.4.4.2 of IEC 61508-3), and planning and management as the initial development of the E/E/PE safety-related systems.
7.8.2.4	After modification, the E/E/PE safety-related systems shall be reverified and revalidated.
Recommended techniques and measures, IEC 61508-3 A.8	
2	Reverify changed software module
3	Reverify affected software modules
4a/4b	Revalidate complete system or Regression validation

perform the re-verification and re-validation of the complete system (measures 2, 3, 4a in the Table 1). Alternatively, regression validation would also suffice (measure 4b). Nevertheless, changed artifacts (from the work products of the hazard and risk analysis down to the test reports) have to be newly certified.

In summary, the change impact on safety integrity implies to update many work products throughout the systems lifecycle, to repeat particular steps of that lifecycle and to re-verify and re-validate the system. However, according to requirements 7.8.2.3 and 7.8.2.5, those activities have to be performed only if there is an impact on the functional safety (i.e., the systems safety integrity is changed). Our goal in this context is to allow changes to an extent to which they have no impact on the systems safety integrity. For this purpose, we need to evaluate the requirement 7.8.2.3-a, for every change request. If there is no need for the hazard and risk analysis, changes are allowed, otherwise not. To realize this, we first need to identify the system attributes that have an impact on systems safety integrity. Based on these attributes, we can set constraints on the architectural level (e.g., software components, layers, operating system configuration, etc.) that would allow us to evaluate the requirement 7.8.2.3-a. In the following, we introduce these attributes.

3.2 Impact of Changes on System Requirements

Safety standards set requirements to achieve the functional safety, while leaving the space for the developers on details on how they should implement those requirements. The same holds for the change management, i.e., the IEC 61508 does not specify which system attributes have to be considered when analysing the impact of changes. More concrete guidelines about this can be found in the avionics domain, concretely in the concept Reusable Software Component (RSC) from the Federal Aviation Administration (FAA) that was developed for the standard

DO-178B, to enable reuse of software components and their late integration into a certified safety-critical system [7]. Similar to change management, the aim is to maintain the functional safety after integrating components. Although RSC provides concrete information about reusing pre-fabricated components, no focus has been given on how to design such components for reuse – for example, how to describe the context in which components have to operate (embedded system, environment, etc.) and which system attributes contribute to that context. Similar to RSC, the concept Safety Element out of Context (SEooC) as part of the automotive standard ISO26262 defines reuse for the sub-systems, but on the abstraction level of requirements.

To our knowledge, the only available official publication that handles change management in detail and is related to safety standards are the FAA guidelines on analyzing the impact of changes in software [6] [17]. Here, a collection of the concrete system attributes that might be affected by changes is presented. This collection is made to help developers in the post-certification process of the DO-178B standard to ensure the safety integrity of the changed system by determining the impact of changes on the system, and by estimating the overhead to re-verify, re-validate and re-certify the system. Although avionics domain is addressed here, most of those attributes are common to embedded systems in general. In Table 2, we summarize the common system attributes.

Table 2. Considered system attributes to analyze impact of changes, according to Federal Administration Aviation (FAA) [6]

System attribute	Description
traceability	requirements, design, tests, procedures
memory margin	memory allocation requirements (volatile, non-volatile memory)
timing margin	timing requirements (task scheduling, interface timing, ...)
data flow	coupling between software components (data syntax, semantics)
control flow	coupling between software components (events, calls, ...)
input/output	interfaces with the external world (bus, hardware, memory, ...)
development environment and process	compilers, linkers, loaders, tools
operational characteristics	runtime mechanisms (changes on limits, i.e. contracts, exception handling, ...)
partitioning	change on protective safety mechanisms

We use some of the FAA attributes as the first class entities to maintain the consistency of the system, and to estimate the impact of changes. We discuss the selection of attributes in the following section more in detail.

4 Ensuring Consistency of System Configurations

In this section, we introduce our approach to ensuring the consistency of system configurations. To this end, we show how we define a system using attributes

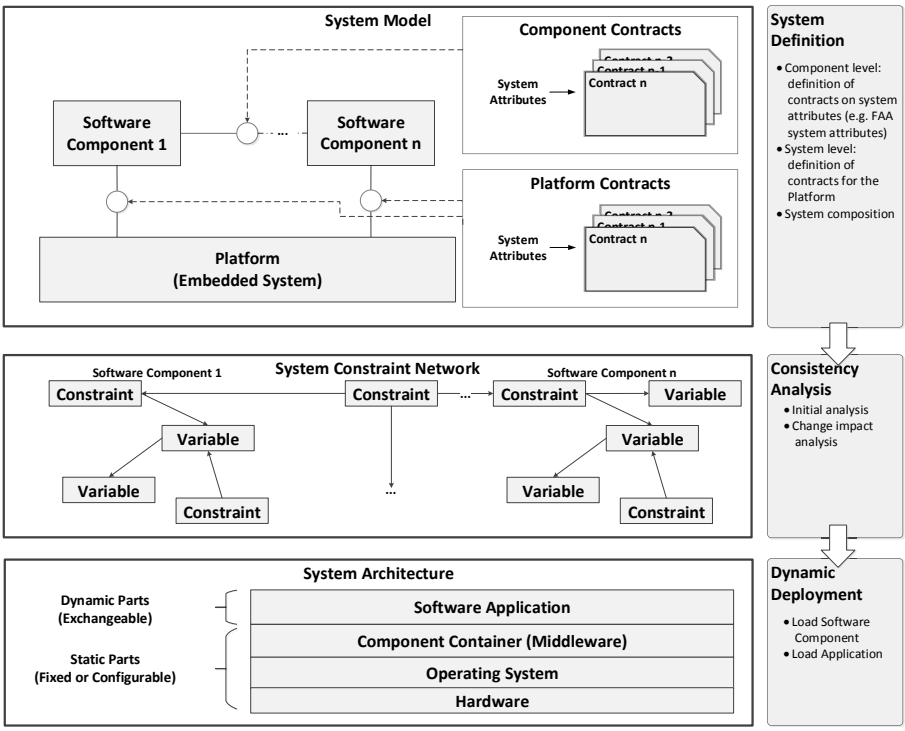

Fig. 1. Proposed workflow for ensuring systems consistency: system modelling using contracts to describe attributes (top), consistency analysis (middle) and dynamic deployment of software components (bottom)

described in the previous section, and how we analyze the impact of changes. All information about systems consistency is contained in those attributes.

The proposed approach in the workflow form is depicted in Figure 1. On the top, a model of the system is defined. This model consists of the two elements: software components which implement certain application-level functions, and the platform, which is a model of an embedded system. Both software components and the platform implement certain contracts, in order to express relations to other dependent components or platform. These contracts are the fundamental elements of the system model that allow us to maintain the consistency of the system. They contain the information about system attributes discussed in the previous section, and provide means to build relationships to other contracts. Based on those relationships, impact of changes in one particular contract can be tracked throughout the complete system. We introduce contracts later in Section 4.2. In the next step of the workflow, the system in terms of contracts is translated into a so called constraint network, i.e., a set of inter-connected variables and constraints. This constraint network represent contracts and their relationships in another problem domain, which allows us to automatically analyze the consistency of the system by evaluating constraints.

In the last step of the workflow, components can be dynamically loaded into the platform, depending on results of the analysis. If all constraints in the analysis step are satisfied, the system modelled in the first step is consistent, i.e., we say the system configuration is assured. Thus, any change in the modelling step can be captured and analyzed in the constraint network.

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

4.1 Software Architecture

To perform changes in the system by replacing some parts of it, there is a need for an adequate architectural support, i.e. a design for upgrades [9]. Another important aspect here is that a degree of flexibility shall be balanced to an extent to which an the impact of changes on system attributes listed in Table 2 can be managed. For example, if changes in behaviour of certain software functions cannot be analyzed (e.g., in the constraint network from Figure 1), replacing those functions with different behaviours shall not be approved. Therefore, certain limitations are necessary to set on the design.

For our system, we use a Component-based Software Engineering (CBSE) [5], which is currently a key paradigm applied for building safety-critical systems. Automotive AUTOSAR, standards such as IEC61131/499 and IEC61850, are some of the reference component-based architectures. In those architectures, software components implement parts of systems functions, such as the controllers, software sensor and actuators. Due to well-defined interfaces, component may implement functions on different granularity levels, e.g., like Matlab Simulink function blocks and sub-systems, thus allowing for compositional (hierarchical) design. Moreover, well-defined interfaces allow their reuse, customization for the use in different contexts, and so forth.

Our software architecture is depicted in Figure 1 (bottom part). Here, software components implement certain software functions composed into an application, whereas their lifecycle, their coordination and resources from the operating system are managed by the underlying middleware, i.e., a component container. Changes related to software may impact any of these layers, and therefore any of the introduced system attributes. In order to be able to analyze such an impact, we set limitation on the design so that replacements of software components are allowed only. That means, some of the system attributes are fixed at design-time so that changes have no influence on those attributes. For example, connections between software components have to be static, since they may affect the functional requirements if changed (e.g., adding/removing software components, or changing connectors may affect systems behaviour), and this can only be analyzed manually.

With our limitations, the impact of changes is related to software components and their interaction with the platform only. However, such cases are also not trivial, since changes may still have an impact on systems consistency. For example, the consistency may be compromised if the replaced software component implements interfaces with different semantics, e.g. different value intervals provided to dependent components, and new intervals were not considered during

the system verification. Similarly, mixing components with different quality levels may cause the same effects, e.g. deploying components qualified for the lower level of the safety integrity than the integrity of the platform.

The main impact of changes here is on (i) resource management, in particular on task and memory management for components, and on (ii) interfaces between components and their interfaces to the platform. From the perspective of the software architecture, the configuration of tasks (i.e., number of tasks, their scheduling policy, etc.) and the organisation of memory (i.e., memory layout, size of the heap, and allocation to tasks) are static features. However, they have to be included in the analysis since exchanged components may have different demands with regard to resources (timing, memory).

Similarly, the connections and interfaces between components are static, but many details have to be considered in order to ensure that the integration or the composition is correct (i.e., the syntax and the semantics, such as units, valid intervals of certain data values, etc.). In addition, some components may implement many alternative behaviours so that different configurations of interfaces are possible. Therefore, we consider interfaces in our analysis. Finally, details with regard to the development process and the operational profile are also parts of the analysis. In Table 3, we summarize system attributes that we include in the analysis, and possible types of changes (right column). The systems consistency is therefore analysed based on these changes only. The remaining systems attributes from Table 2 are fixed at design time, and cannot be influenced, i.e., the control flow of components, their interaction semantics and behaviour are implemented as a static part of the architecture.

Table 3. System attributes considered in the consistency analysis, selected from FAA attribute collection [6]

System attributes	Allowed changes
memory margin	components: volatile and non-volatile memory platform: volatile memory (allocated to tasks), non-volatile memory
timing margin	components: execution time platform: task execution time
data flow	components/platform: syntax (datatype, interface), semantics (intervals, values, specific constraints (units, standard compliance, configuration and calibration data, ...))
development environment and process	components/platform: tools (compiler, linker, specific constraints (build options, ...)), version
execution platform	components/platform: architecture (cpu, floating point support, ...), safety integrity level

4.2 System Modelling

To integrate the information about selected attributes from Table 3 into the systems structure, we use Contract-based Design paradigm (CBD) [3]. According to CBD, software components and the platform implement certain contracts, which capture part of that information (i.e., a quality stamps or properties). In addition to capturing information, contracts provide means to integrate components and platform.

Among few types of different contract available, such as state transition-based contracts like interface automata, probabilistic contracts, etc., we use a form that is based on data semantics, i.e., data flow contracts [3]. According to this type, every contract consists of data parameters, and expressions (or properties) on those data parameters in form of assumption and guarantees. The guarantees describe a valid behaviour in form of expressions, that can evaluate to true/-false, depending on the evaluation of expressions in assumptions. For specifying contracts, various formalisations can be used, for example logic languages such as propositional logic, first order logic, their extensions, and other.

In Figure 2, we show how the structure of our software components is defined to match with used contracts, and how different types of system attributes are modelled using contracts. A trivial example is shown here just to simplify the demonstration. Similar to the structure of contracts, every software component and the platform are defined as a set of data parameters, input and output data variables. In addition to this information, they contain a list of implemented contracts. Thus, data parameters in contracts relate to data parameters of their corresponding components/platform.

Another essential aspect of CBD are the relationships between contracts, which allow to verify the composition between two contracts, if their assumptions and guarantees are defined in a formal way. In our example, the contracts of components M_{IS} and M_{IIAS} are related with each other using a composition relation, which is valid only if these contracts are compatible and can interact. Concretely, this means the relation is valid if the contract of a component which accepts data, the $C1$ implemented by component M_{IIAS}, can be satisfied by

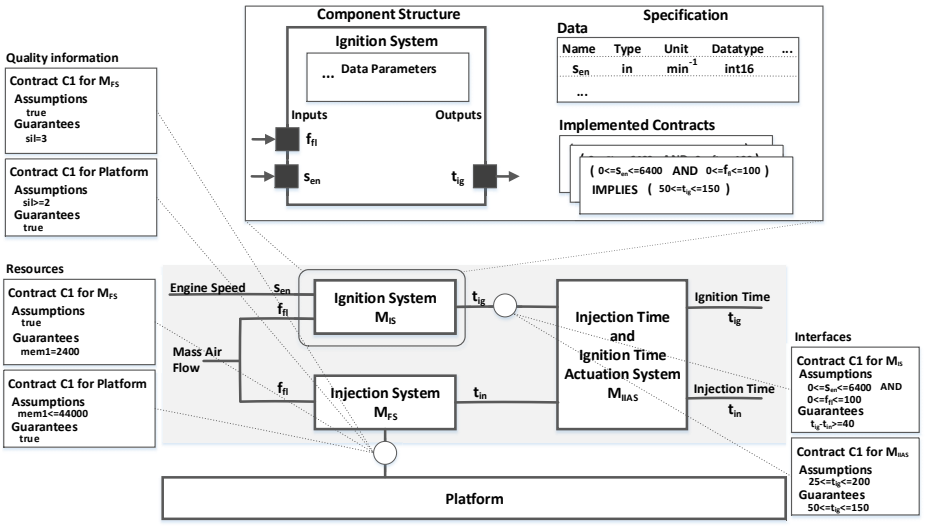

Fig. 2. Structure definition of software components and the platform, and supported types of contracts shown on an example of the engine controller, adopted from [8]

the guarantees of the providing component. In the example, this can evaluate to true only if the assumption ($25 \leq t_{ig} \leq 200$) can also evaluate to true[2]. This can only be satisfied, if the guarantee of the contract $C1$ of M_{IS}, ($50 \leq t_{ig} \leq 150$), matches with the assumption ($25 \leq t_{ig} \leq 200$), which is the case in the example, since M_{IIAS} can accept more values of t_{ig} (for more details, please refer to [11]). Therefore, guarantees and assumptions of dependent components are interrelated by their expressions. In a similar way, we define contracts for resources and quality information, as shown in the figure.

Based on relationships between contracts, information about the system attributes on a system level is maintained. Changes in any contract (or exchanges of contracts) can be captured by evaluating assumptions/guarantees of other dependent contracts.

4.3 Consistency Analysis

The consistency analysis is based on verifying relations between contracts, in particular, by evaluating their assumptions and guarantees. As a background technology, we use Constraint Programming paradigm (CP) [14], which is a widely applied method to solve decision and optimization problems. The essential aspect of CP is a problem definition, which is represented as a network of variables of various types and constraints. Here, constraints represent various kinds expressions on variables (logical, arithmenic, etc.), and can be related to other constraints. Solving that problem means evaluating all constraints in the network. Thus, there is a solution if all constraints in the network are satisfied.

In our approach, we translate the system modelled in form of contracts into such a constraint network. For this purpose, we have defined a model of a contract, its variables, assumptions and guarantees, and relations between contracts as network elements, i.e., variables and constraints. The systems consistency is therefore analyzed by evaluating constraints that are derived from contracts (for more details, please refer to [11]).

5 Discussion

We showed in this paper that simple replacements of software components are not trivial. Many details about functional and non-functional aspects of software components have to be considered to ensure that replacements have no impact on systems integrity. One of the major challenges here is to determine how much information should be considered in the analysis, to have a confidence in its results. With the list of systems attributes we introduced in this work, some fundamental aspects are covered, but much more details might be required, depending on the specific domain. This collection of attributes can be extended according to types of contracts introduced.

[2] This data is related to the ignition time of an engine controller t_{ig}. The components modelled here implement contracts in order to satisfy the timing requirement on allowed difference between injection and ignition time, i.e. ($t_{ig} - t_{in}$).

The analysis method presented here can also be applied to some existing component-based systems, but in some cases with certain limitations. In AUTOSAR for example, changes would be possible on a level of Runnables, which are units of the execution inside of AUTOSAR software components, and have a generalized standard behaviour (read, execute, write) [12]. In contrast, changes of complete software components could not be supported, because events for the execution of Runnables are user-defined, and other techniques are required here to analyze the interaction between those events. Generally, for synchronous data flow systems, such as IEC61131-based systems, or Matlab Simulink function blocks, it is more easily to apply the analysis, since software components used here have a standard behaviour and standard execution semantics.

6 Conclusion

In this paper, we presented an approach to perform changes on safety-critical embedded systems in the operation and maintenance phase. Changes are limited to replacements of software components. To prevent the impact of such type of changes on systems integrity, we have analyzed which related system attributes might be affected when replacing software components. Based on those attributes, we provided a modelling means to build a system including attributes on the level of software components and their platform (embedded system), and we provided a consistency analysis of such a modelled system. The main outcome of this work is that for replacements of software components the system does not need to be turned back into the development phase.

The collection of attributes described here provides a foundation for the future work. One of the major challenges here is to determine how much information is required to describe software components and their platform, in order to have a confidence on results of the consistency analysis. This may depend on many factors, such as used software architecture, domain-specific details, and so forth.

As part of our ongoing work, we will analyse different component-based architectures with regard to the use case of replacing software components, and derive specific system attribute out of them. The aim is to extend the proposed modelling and analysis support to system attributes, which can be commonly used in safety domains.

References

1. Adler, R., Schaefer, I., Trapp, M., Poetzsch-Heffter, A.: Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. ACM Trans. Embed. Comput. Syst. 10(2), 20:1–20:39 (2011)
2. Alemzadeh, H., Iyer, R., Kalbarczyk, Z., Raman, J.: Analysis of safety-critical computer failures in medical devices. IEEE Security Privacy 11(4), 14–26 (2013)
3. 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, 2012, Inria (2012)

4. Butz, H.: Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. Department of Avionic Systems at Airbus Deutschland GmbH Kreetslag 10, D-21129 Hamburg, Germany (-)
5. Crnkovic, I.: Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood (2002)
6. FAA: Guidelines for the Oversight of Software Change Impact Analyses used to Classify Software Changes as Major or Minor. Notice 8110.85, FAA (2000)
7. FAA: AC20-148 Reusable Software Components. Tr, FAA (2004)
8. Frey, P.: Case Study: Engine Control Application. Tech. rep., Ulmer Informatik-Berichte, Nr. 2010-03 (2010)
9. Gluch, D., Weinstock, C.: Workshop on the State of the Practice in Dependably Upgrading Critical Systems: April 16-17, 1997. Special report, Carnegie Mellon University, Software Engineering Institute (1997)
10. Kajtazovic, N., Preschern, C., Kreiner, C.: A component-based dynamic link support for safety-critical embedded systems. In: 20th IEEE ECBS (2013)
11. Kajtazovic, N., Preschern, A., Hoeller, C., Kreiner, C.: Constraint-based verification of compositions in safety-critical component-based systems. In: IEEE/ACIS SNPD (June 2014)
12. Kindel, O., Friedrich, M.: Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis. dpunkt Verlag, Auflage: 1 (2009)
13. Lopez-Jaquero, V., Montero, F., Navarro, E., Esparcia, A., Catal'n, J.: Supporting arinc 653-based dynamic reconfiguration. In: 2012 Joint WICSA and ECSA (2012)
14. Marriott, K., Stuckey, P.J.: Programming with Constraints: An Introduction. The MIT Press (March 1998)
15. Montano, G.: Dynamic reconfiguration of safety-critical systems: Automation and human involvement. PhD Thesis (2011)
16. Pop, P., Tsiopoulos, L., Voss, S., Slotosch, O., Ficek, C., Nyman, U., Ruiz, A.: Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the recomp approach. In: WICERT (2013)
17. Rierson, L.: A systematic process for changing safety-critical software. In: Proceedings of the 19th Digital Avionics Systems Conference, DASC, vol. 1, pp. 1B1/1–1B1/7 (2000)
18. Smith, D., Simpson, K.: A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849. Elsevier Science (2010)
19. Soliman, D., Thramboulidis, K., Frey, G.: A methodology to upgrade legacy industrial systems to meet safety regulations. In: 2011 3rd International Workshop on Dependable Control of Discrete Systems (DCDS), pp. 141–147 (June 2011)
20. Zhang, M., Ogata, K., Futatsugi, K.: Formalization and verification of behavioral correctness of dynamic software updates. Electronic Notes in Theoretical Computer Science 294, 12–23 (2013); Proceedings of the 2013 VSSE Workshop