A Compositional Safety Specification Using a Contract-Based Design Methodology

Markus Oertel*, Peter Battram†,
Omar Kacimi‡ and Sebastian Gerwinn§
OFFIS – Institute for Information Technology,
Escherweg 2, 26121 Oldenburg, Germany
Email: *oertel@offis.de, †battram@offis.de,
‡kacimi@offis.de, §gerwinn@offis.de

Achim Rettberg
Carl von Ossietzky University Oldenburg
Escherweg 2, 26121 Oldenburg, Germany
Email: rettberg@iess.org

***Abstract*—Model-based design methodologies have become the standard approach to develop safety critical systems. Therefore, many approaches exist to model faults, failures and their propagation. Nevertheless, due to the frequent use of off-the-shelf components as well as the need to react efficiently on changes, the importance of modular and compositional techniques is gaining constantly. Here, we present an approach for compositional reasoning on safety specifications that supports multiple abstraction levels in the design process. Especially in the safety domain, it is obvious that a safety concept is just valid under certain conditions, e.g. that only a limited amount of components may fail at the same time. Therefore, we extend existing safety specification methods based on contracts, which explicitly distinguish between assumptions and guarantees, building a well-founded framework for compositional reasoning. Our formalization method can be used to develop a safety specification starting from the top level system component and refine it until the lower hardware and software layers while preserving the validity of early performed analyzes. On a practical level, we further describe how safety specifications can be formalized into a model checking problem and analyzed using existing tools.**

***Keywords*–Safety Critical Systems, Safety Contracts, Contract-based Design, Model-based Design, Fault Modeling, Model Checking, Formal Methods**

I. INTRODUCTION

Safety relevant systems are characterized by a high amount of required verification and validation activities necessary for qualification or certification. Changes in the system often cause a tremendous re-verification effort. Fenn et al. [1] described the experience of industrial partners in which the costs for re-certification are related to the size of the system and not to the size of the change. Also, Espinoza et al. [2] discovered that the monolitic and process oriented structure of the safety cases required by nearly all domain-specific safety standards may require an entire re-certification of the system after changes. Therefore, recent work has focused on establishing modular specifications and processes aiming at preserving the safety properties even in the presence of changes [3][4][5]. Additionally, from a computational perspective, modular and decomposable specifications are highly desirable as the overall computational load can also be substantially reduced. That is, the analysis of the whole system in one state-space may not be feasible, while those of sub-parts are expected to be still in an acceptable performance corridor.

To obtain a compositional safety aspect, a specification language is needed that can describe the fault propagation and mitigation of a component, as well as a theory for composing components and comparing their specifications. Existing approaches to specify safety properties in a modular way (see Section II for an overview) either lack expressiveness, or compositionality as defined by Hungar [6] and Peng [7]: there exists a separation between the specification of a component and its actual behavior. If a component is replaced by another component meeting its original specification, the correct functional behavior of the composed system is maintained.

The main contribution of this paper is to add support of abstraction techniques to safety specifications to enable a top down design process. Requirements on higher abstraction levels leave room for many implementation possibilities (therefore also many different fault propagations), e.g. requiring the absence of single point of failures, which are specialized on lower levels of abstraction. The correct break-down of requirements can be automatically analyzed. The benefit of our specification is that the analysis results on higher level are not compromised if the system is further refined. Another benefit of such an approach is the possibility to automatically compare the assumed fault propagation with the behavior implemented [8]. Our work is based on the *design by contract* principle, which has a well-founded semantic framework for composing components and argue on the correctness of their refinement. As a language, we extend the already existing safety contracts with compositional aspects.

The paper is organized as follows. We briefly review the contract-based approach in Section III together with the existing safety contracts. We describe in Section IV how we can abstract from concrete component failures on higher abstraction levels and we propose a mapping to a linear temporal logic (LTL) based model checking problem (see Section V). Finally, we apply the approach to an example use-case (Section VI) and conclude our work in Section VII.

II. RELATED WORK

We briefly present the already existing approaches, which provide a modular or hierarchical safety specification for

 ISBN: 978-1-61208-401-5

embedded systems.

A. HipHops

Hip-Hops [9] is a failure logic modeling-based approach [10] that aims at automating the generation of safety analysis artifacts such as Fault Trees and Failure Mode and Effect Analysis tables (FMEA). To this end, design models, e.g., SIMULINK models [11], are annotated with a fault propagation specification, which states for a component possible input and internal faults as well as how they influence the output. The deviations from the nominal behavior considered by Hip-Hops follow HAZOP [12] guidelines and include, for example, HAZOP guidewords *omission*, *too late*, and *value*. The atomic faults can be annotated with probabilities of occurrence, which in turn can be used along the structure of the generated fault tree to calculate the overall probability of a failure at top level. The popularity of the approach is based on the seamless integration in existing safety processes, since automation support is added to well-known analyses.

Our approach is more requirement-oriented. We directly formalize safety requirements and address the correctness of their refinement. Although the stated fault propagation relationships are similar, there is no need to generate a set of fault trees for each failure and their manual interpretation. As we are focusing on safety requirements and proving the correctness of their break-down, we do not need to compare fault trees with requirements.

B. Fault Propagation and Transformation Calculus

The fault propagation and transformation calculus (FPTC) [13] is addressing the problem of a modular safety specification. The architectural design is expressed using Real-Time Networks (RTN) to model the communication channels. The calculus itself is based on direct propagation notations using failure modes in accordance to HAZOP guidewords. The approach claims to be modular or compositional by providing a fixed-point algorithm to calculate the set of all possible occurring input and output faults for every component in the system. In contrast to the solution presented in this paper, the FPTC approach does not provide any support for multiple abstraction levels, i.e., specifying a component, then designing the subcomponents, and prove the correctness of the refinement.

Additionally, within FPTC it is not possible to state temporal properties in the propagation of faults (e.g. that a fault is going to be detected after some time). Furthermore, in case of multiple input and output ports, the notation of *positive propagation* is more complicated than the *negative fault containment* properties stated within safety contracts. Also, the order of the evaluation of the FPTC requirements matters, which is not the case for the invariant safety patterns. A minor drawback of FPTC is also the necessity of an RTN model, while safety contracts can be attached to any port-based component model.

C. Cause Effect Graphs

Kaiser et al. [14] address the problem that the typical structure of a fault tree is related to the hierarchy of the influences of a failure, but not to the hierarchy of the system. Therefore, they extend fault trees to be directed acyclic graphs, which they call Cause Effect Graphs (CEGs). This representation has the benefit of uniquely representing repeated events (identical causes used in multiple sub-trees) in fault trees. The CEGs allow to create a direct mapping of parts of the fault tree to components in the architecture, making these elements re-usable. A tool called UWG3 has been developed to evaluate the approach.

Kaiser et al. address compositionality and re-useability in their approach, but do not integrate abstraction. At the time of the analysis the model still needs to be complete and cannot be refined later without the need of a full re-evaluation. The same limitations with respect to the interpretation of fault trees as they have been stated for Hip-Hops (see Section II-A) also apply to this approach.

D. Previous work on safety contracts

Boede et al. [15] presented an approach for hierarchical safety specifications using safety contracts. They introduced patterns to describe the hierarchy of faults, the hierarchy of functions, and the relation between faults and functions, e.g.: `function <function-name> can be impacted by <failure-list>`. Although the approach is based on contracts, the stated patterns do not fully exploit the contract notation as assumptions are neglected, a major principle of contracts.

While stating the need for a hierarchical failure specification language, the semantics for the presented patterns are not precisely given. Hence, it is hard to estimate if the proposed analysis can be automated.

Oertel et. al [16] presented an approach for expressing safety properties in a contract-based fashion providing formally defined semantics and application guidelines. The presented approach, however, did not consider abstraction techniques to refine safety concepts suitable for a top-down design. Since the approach presented in this paper is extending the work of Oertel et. al [16], the main contract templates and their semantics are briefly presented in Section III-B.

III. Background

A. Overview Contract-based Approach

Contracts [17][18] separate a requirement into an *assumption*, which describes the expected properties of the environment, and a *guarantee*, which describes the desired behavior of the component under analysis, provided the assumption is met by the operational context, i.e., the environment. This separation allows for building a sound theory thereby permitting to reason in a formal way about the composition of systems. Contracts, belonging to the class of assume-guarantee reasoning techniques [19], are a widely adopted approach for compositional verification [20][7].

Contract semantics for reactive and embedded systems are defined over traces of systems [6][18]. Components, in the following denoted with $\mathtt{M}$, are characterized by ports (P), that are either defined as input or as output ports. A trace assigns a value V out of the value domain $\mathcal{V}$ to each of the ports at any

given point in time $t \in \mathcal{T}$. Therefore, a trace is of the form $[\mathcal{T} \rightarrow [P \rightarrow \mathcal{V}]]$. The traces of a component M are denoted $[\![\mathtt{M}]\!]$. This set comprises all possible behaviors of a component (implementation), even in case of unacceptable inputs due to a general requirement of input openness, i.e., requiring systems to never confine or otherwise refuse input. The (semantically concurrent) composition of multiple sub-components $\mathtt{M}_1 \ldots \mathtt{M}_n$ to a component M is defined [18] as the set of traces accepted by all components:

$$[\![\mathtt{M}]\!] = \left[\!\!\left[\bigtimes_{i=1}^{n} \mathtt{M}_i \right]\!\!\right] = \bigcap_{i=1}^{n} [\![\mathtt{M}_i]\!]$$

In some modelling languages it is possible to name the port-ends of delegation- and assembly-connectors differently. For the same of simplicity we do not foresee such a behavior and assume to have identical port names if they are connected.

A contract is a tuple $C = (A, G)$ describing a set of traces using the assumption A and the guarantee G:

$$[\![C]\!] = [\![A]\!]^{-1} \cup [\![G]\!]$$

with $(.)^{-1}$ denoting the complement of a set. Although not formally required by the definition of contracts, assumptions typically only specify constraints on the input, whereas guarantees reflect the input/output relation and therefore contain restrictions of allowed outputs while being input-open. We assume all contracts to be stated in the *canonical form* [21], which allows some simplifications in the following definitions of the operators and relations. A contract is said to be in canonical form if $[\![G]\!]$ at least contains $[\![A]\!]^{-1}$.

In order to reason about the correctness of the composition of a system, a set of basic relations and operators are defined on contracts and implementations.

Definition 1 (Satisfaction)**.** *A component M satisfies [18][22][21] a contract $C = (A, G)$, denoted $M \models (A, G)$, iff all its traces are permitted by the contract, i.e., $[\![M]\!] \subseteq [\![C]\!]$, and its inputs and outputs coincide to those underlying the contract. Consequently,*

$$M \models (A, G) \Leftrightarrow [\![M]\!] \cap [\![A]\!] \subseteq [\![G]\!]$$

Refinement is a relation between two contracts, stating that the refined contract is a valid concretisation of the other, i.e., the refining contract is a valid replacement in all possible operational contexts satisfying all (and maybe more) requirements satisfied by the refined contract.

Definition 2 (Refinement)**.** *According to [17] and [22] a contract C_1 refines C_2 iff it has the same signature, i.e., same input and output ports, yet imposes relaxed assumptions and more precise guarantees:*

$$C_1 \preceq C_2 \Leftrightarrow A_1 \supseteq A_2 \wedge G_1 \subseteq G_2$$

Parallel composition ($\otimes$) is used to combine multiple contracts into a new contract that represents the behavior of (concurrently) composed components each of which satisfies their individual contracts.

Definition 3 (Parallel Composition)**.** *For two contracts C_1 and C_2, the parallel composition $C_1 \otimes C_2$ is defined as:*

$$((A_1 \cap A_2) \cup \neg(G_1 \cap G_2), (G_1 \cap G_2))$$

B. Overview of Safety Contracts

The assumption and the guarantee of a contract can be described using various languages depending on the analysis target. In case of safety requirements we use a formal, pattern-based language, called *safety patterns* [16], which can be translated to many other suitable target languages, such as LTL or timed-automata [18]. By using LTL in [16], four safety patterns are introduced to express different safety properties like fault propagation, safety mechanisms and transitions to a safe-state. These patterns are depicted in TABLE I.

TABLE I. Overview of the Safety Patterns

Pat. 1	Structure	**none of** $\{\{m_1, m_2\}, \{m_3, m_4\}\}$ **occurs**
	Intuition	This pattern speaks about malfunction or mode (m) combinations that are not allowed.
	LTL expression	$(\mathbf{G}\neg m_1 \vee \mathbf{G}\neg m_2) \wedge (\mathbf{G}\neg m_3 \vee \mathbf{G}\neg m_4)$.
Pat. 2	Structure	expr-set **does not occur**.
	Meaning	Derived from pattern 1, where just one set of malfunctions or modes is used. Semantics are identical. Introduced for convenience.
Pat. 3	Structure	m_1 **only followed by** m_2
	Intuition	It is proposed to restrict the occurrence of m_1. The pattern is typically used to define consecutive modes.
	LTL expression	$\mathbf{G}(m_1 \rightarrow (m_1 \mathbf{W} m_2))$.
Pat. 4	Structure	m_1 **only after** m_2
	Intuition	It is proposed to restrict the occurrences of malfunctions or modes. This pattern is typically used to express that a safety mechanism shall only be activated after the fault occurrence.
	LTL expression	$\neg m_1 \wedge G((\mathbf{X} m_1) \rightarrow (m_1 \vee m_2))$

Pattern 1 and its derived Pattern 2 state that only system runs (traces) are accepted in which the combination of malfunctions (as stated in the expression sets) is absent. Pattern 3 and Pattern 4 are used to order malfunctions or modes of the system.

Oertel et. al [16] presented several contract templates that proved to be useful for automotive safety concepts. We are focusing here on a subset of them, since the others can be handled identically. The basic template is depicted in contract C_1 and describes the robustness of the output of a component against internal faults. More precisely, the contract explicitly mentions all combinations M_f of malfunctions on the inputs (Inp_{mf}) or internal malfunctions (Int_{mf}) of an component that could cause the output to fail (`out_fail`). Therefore, $M_f \subseteq \mathcal{P}(Inp_f \cup Int_f)$ and the corresponding safety contract consisting of assumptions A and guarantees G can be written as:

$$C_1 \quad \begin{array}{l|l} \mathbf{A}: & \mathbf{none\ of}\ \{M_f\}\ \mathbf{occurs.} \\ \mathbf{G}: & \{\mathtt{out\_fail}\}\ \mathbf{does\ not\ occur.} \end{array}$$

In order to fulfill the requirements of the ISO 26262, safety contracts are able to express the degradation of a system, i.e., switching to a safe-state. This safe-state is expressed in functional terms (e.g., functional variables need to be in a defined range), and is therefore just considered as an identifier

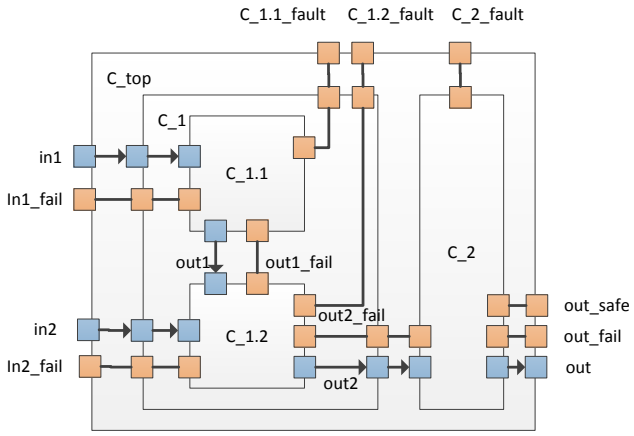

Figure 1. All internal faults were "externalized" using fault activation ports. Blue ports are considered as functional ports, while red ports describe the failure modes of the functional ports.

in safety contracts. Contract template C_2 is using this mechanisms to state that a system is either operating normally or that it is in a safe-state, if the assumption is met. The assumption is violated if one of the stated combinations of malfunctions occurs.

$$C_2 \quad \begin{array}{l|l} \textbf{A:} & \textbf{none of } \{M_f\} \textbf{ occurs.} \\ \textbf{G:} & \{\texttt{out\_fail } \textbf{and } \texttt{!out\_safe}\} \textbf{ does not occur.} \end{array}$$

In Oertel et al. [16], the top level contract talks about all non-desired atomic fault combinations, even if the atomic faults correspond to components of a more detailed abstraction level. Figure 1 illustrates that all faults in the most detailed (lowest abstraction level) components need to be made available to the containing components by use of fault ports. I.e., that the designer needs to know all possible internal faults while specifing the top-level component. This approach is not compositional, since changes on internal components directly cause changes on the containing component. Furthermore, a top-down design is impossible, since it is assumed that all components and their faults are known in advance.

IV. Compositional Faults & Fault Abstraction

Every component is only safe under certain conditions. For example, a dual-channel architecture is only safe as long as at least one channel is working correctly. In a contract-based approach these assumptions are explicitly stated for each requirement. Therefore, the contract assigned to the top level component guarantees the correctness or "safeness" of an output while assuming only a limited set of combinations of particular internal faults to occur in the system.

The assumptions that restrict the correctness of an output signal can in many cases be abstracted to restrict only the number of faults within the system. Using this technique we are still able to express the absence of single-points of failure, as required by the ISO 26262, but do not need to state all particular faults individually. Thereby this generalization enables a top-down design approach.

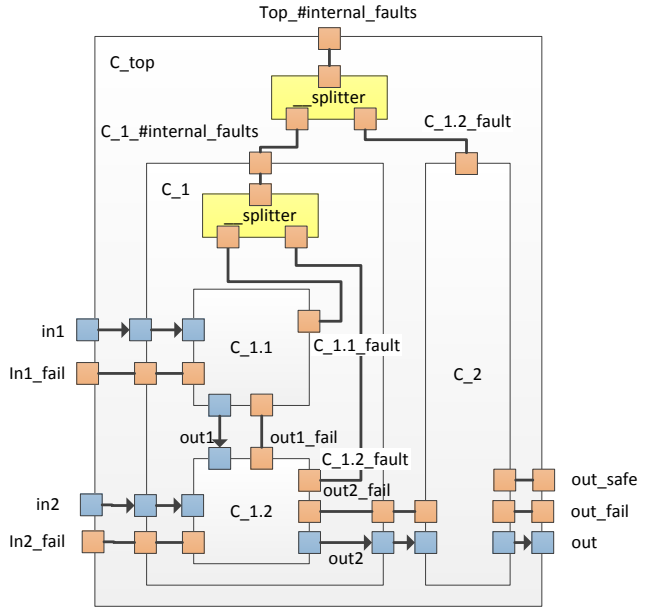

Figure 2. Count ports are introduced and fault-splitter components, that ensure that not more faults are passed to internal components as specified

Therefore, since traces are based on values on ports, we introduce *fault count* ports representing how many internal faults are present in a component (see Figure 2). Again, blue ports represent actual functional values that are used in the implementation of that component and red ports represent failure modes assigned to these ports. Notice, that more than one failure mode can be assigned to a functional port, as it is the case the `out` port. Using fault count ports, only at the most detailed abstraction level, i.e., the implementation, the particular faults that might occur in a component need to be specified. The fault count ports are of type Integer and belong to the safety aspect of a system only. From a functional point of view, these ports are virtual and do not occur in any generated code. Additionally, we also introduce virtual *splitter components* to create a connection between the fault count port on the higher component and the fault ports (or also fault count ports) on the sublevel components. They are called virtual, since they are not necessary for the specification, they are just necessary to still be compliant to the contract based design methodology and can therefore be generated automatically. The splitters distribute the maximum number of faults of the surrounding component to its subcomponents. Consequently, they allow multiple possibilities of how to distribute the number of allowed faults. On the one extreme, they could be just assigned to one single component and all the other subcomponents are assumed to be fault free or, on the other extreme, they could be evenly distributed aross all subcomponents. Under all of these possibilities the component needs to be safe. In terms of trace semantics, these splitters only allow traces that do respect the number of faults. For n subcomponents, the contract associated to the splitter component can also be characterized by the following contract:

$$C_3 \quad \begin{array}{l|l} \textbf{A:} & \texttt{true} \\ \textbf{G:} & \texttt{out}_1 \texttt{ + ... + out}_n \texttt{ < \#internal\_faults} \end{array}$$

The guarantee creates a relation between the input port of the splitter (#internal_faults) and the output ports (out_n). If the assumption of a component states that x internal faults do not occur, this contract ensures that only less than x faults are activated on the subcomponents.

As the components are still characterized by input and output ports contracts are still applicable to the safety aspect using traces (see Section III) on fault-count ports and all definitions and analyses from contracts (refinement, etc.) remain valid. Fault count ports are only used for the internal faults of an component, fault that occur at the input ports are not affected since these faults can be easily passed to an additional subcomponents, which are introduced in a refinement step, without violating compositionality.

Intermediate and top level component's safety requirements can be specified using these counting fault ports. The most common usage for the top level assumption is to state that only one or two faults occur in the system. For example, a specification for the component C_1 in Figure 2 could be stated as follows:

C_4

A: `none of {{in1_fail, in2_fail},`
`{C_1_#internal_faults=1, in1_fail},`
`{C_1_#internal_faults=1, in2_fail},`
`{C_1_#internal_faults=2} } occurs.`

G: `{out2_fail AND !out2_safe} does not occur.`

This specification defines that the component shall be robust against one arbitrary fault, which may occur at one of the inputs or internally. Another useful scenario is, that the inputs are assumed to be correct and only internal faults are considered. Such a contract can be stated as follows:

C_5

A: `none of {{C_1_#internal_faults=2},`
`{in1_fail}, {in2_fail} } occurs.`

G: `{out2_fail AND !out2_safe} does not occur.`

The specification of an atomic component (like C_1.2 in Figure 2), for which the internal faults are known, uses the known internal faults, since no abstraction by using a fault count is necessary. The internal faults are represented as individual fault ports (see C_1.2_fault on component C_1.2 in Figure 2) and are directly connected to the splitter components. This ensures, that the specification of the atomic component can still be done accordingly to the guidelines presented by Oertel et al. [16], however one can additionally restrict the total number of different individual faults occurring simultaneously (in the same trace) on a more abstract level.

V. Semantics and Verification of Fault Abstraction

To be able to automatically check our specification and to provide precise semantics, we provide a mapping to boolean LTL. [23]. This decision is based on the availability of fast LTL model checkers and the benefit of relying on the same formalism used for the safety patterns itself (see Section III-B). Furthermore, assumption including an internal faults count bigger than 3 is not expected. Therefore, we cannot use the fault-count ports as integers, but need to create a mapping to boolean logic. Still, from a users perspective the workflow does not change, the specification should be written using the integer fault ports, the mapping is performed completely automatically.

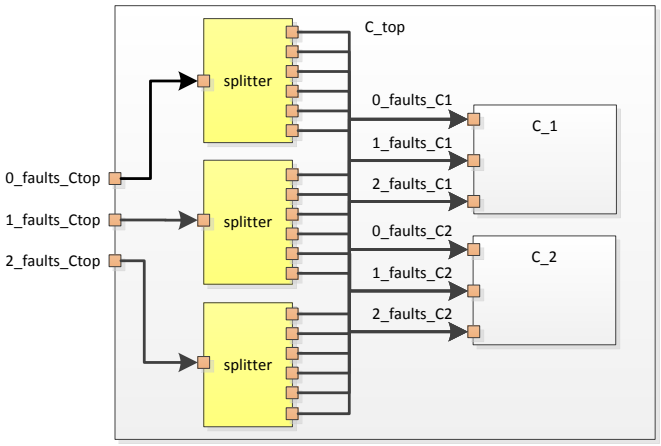

Figure 3. Explicitly represent the counting with boolean logic for LTL implementation. In this example a fault port with the value 2 is represented.

Since all numbers are explicitly stated, and we do not consider unbounded variables, the integer ports can be expressed in a combinatorial fashion. Hence, a fault count port is split up in individual boolean ports (*explicit fault count ports*) representing each a valid value of this port (see Figure 3). If a contract assumes that not more than n faults occur, all numbers of faults smaller than n are valid values. It needs to be stated, that only one of these ports can be active at a time. This is an assumption that needs to be added to all contracts that are attached to the component, e.g., for C_{top}:

C_6

A: `none of {`
`{0_faults_Ctop, 1_faults_Ctop},`
`{1_faults_Ctop, 2_faults_Ctop},`
`{0_faults_Ctop, 2_faults_Ctop}`
`} occurs.`

G: `...`

Furthermore, we need a separate splitter component for each explicit-fault-number port. These splitters allow all logical combinations of ports that sum up to the defined number of faults. For n internal faults and m subcomponents the splitter contract restricts the set of all possible occurring malfunctions $M = \mathcal{P}(\{X_0\_faults\_C0, \ldots, X_n\_faults\_Cm\})$ to M_f:

C_7 **A:** `true`

G: `none of {`M_f` } occurs.`

with $M_f = \{\{x_i\_faults\_C_y\} \subseteq M : \sum_{i=1}^{(n+1)\cdot m} x_i \geq n\}$.

For the purpose of a single refinement analysis we can simplify the translation, since at analysis time splitters are not needed and can be replaced by all combinations of faults of the subcomponents directly. For a given set of internal faults I, the fault-count port `#internal_faults=n` in the assumption resolves to all sets of faults of length n of the powerset $\mathcal{P}(I)$. The refinement check is then implemented as

 ISBN: 978-1-61208-401-5

a satisfaction check on the LTL properties of the contracts (see Section III-A). Rozier and Vardi [24], as well as Li et al. [25] suggested to use a generic model, allowing all possible behavior, to check the property against to reduce the problem to a classical model checking problem.

VI. EXAMPLE

We model a fail-safe temperature sensor as an example (see Figure 4) to apply the top-down design process for safety contracts. The main safety requirement for the temperature sensor is to still deliver a correct or at least *safe* result under the assumption that there is at most one fault present within the system:

C_8

A: `none of {{#sensor_faults=2}} occurs.`
G: `{temp_out_fail AND !temp_out_safe} does not occur.`

The safe-state of the temperature sensor is to output the maximum temperature. This is a valid safe state if the temperature sensor is used in a cooling system, where overheating should be prevented.

A realization of this requirement can be a design with two simple but redundant temperature sensors in addition to a logic, which provides a voting mechanism. The sensors themselves do not provide any safety mechanisms and hence can fail immediately as a result of a single internal failure. Therefore, the safety contract for the temperature sensors 1 is (temperature Sensor 2 is specified in an identical manner):

C_9

A: `none of {{temp1_fail}} occurs.`
G: `{temp1_out_fail} does not occur.`

The logic component shall react to faults of the temperature. As the internal structure has not yet been decided, the requirement states, that even if one of the inputs fails or an internal fault occurs the result should at least be *safe*. Such a requirement can be expressed using a safety contract:

C_{10}

A:
```
none of {
{temp1_out_fail, temp2_out_fail},
{#logic_faults=1, temp1_out_fail},
{#logic_faults=1, temp2_out_fail},
{#logic_faults=2}
} occurs.
```
G: `{temp_out_fail AND !temp_out_safe} does not occur.`

In the refinement step of the logic component, two independent analog/digital converters are used to digitize the temperature signal. Both converters do not provide safety mechanisms and fail immediately:

C_{11}

A: `none of {{ad1_fail}} occurs.`
G: `{ad1_out_fail} does not occur.`

Again, the second converter is specified similarly. The signals are processed by an overwrite component, which compares the results and sets the safe-state if the values differ. The override component is not expected to fail in the lifetime of the device. It is a common assumption in voting architectures to assign a very small functionality to a component that is formally verified and produced in a more robust way than the rest of the system or replaced in a regular manner during service intervals. The safety contract is therefore specified only in terms of input faults:

C_{12}

A:
```
none of {
{ad1_out_fail, ad2_out_fail}
} occurs.
```
G: `{temp_out_fail AND !temp_out_safe} does not occur.`

The corresponding splitter component, the connection of the splitter to the faults of the subcomponents as well as the contract for the splitter component can be generated automatically and do not need to be specified separately.

Refinement can now be checked on both levels of abstraction.

$$(C_7 \otimes C_8 \otimes C_9) \preceq C_6$$

as well as

$$(C_{10} \otimes C_{11} \otimes C_{12}) \preceq C_9$$

VII. CONCLUSION

Since the verification and validation of safety-critical systems consume up to 40% of the development costs, it is even highly undesirable if changes in a system require a re-verification of the whole system. Modular safety cases are expected to reduce these costs by enabling a determination of the area of the system affected by the incorporated changes. In this paper, we presented an approach to enable black-box safety specifications using safety contracts. In contrast to other approaches contracts provide a means of abstraction thereby allowing to develop a system in a top-down manner, i.e., to refine the specification by introducing the possible architecture of the sub-components. Being able to analyze the correctness of a refinement in an LTL-based model checking tool, we are now able to determine if a change in safety requirements needs further adaptations of the system to be compliant to that change. Although this approach is expressive in its specification, it is still intuitively to use.

The currently used LTL-backend does not allow to analyze big system. For some examples with complex specifications, we were not able to check more than 10 contracts in one refinement analysis. Alternatively, an automaton-based representation together with a bounded model checking technique could be a promising candidate for improving the efficiency. Furthermore, it seems to be possible to annotate the faults with probabilities of their occurrence to extend the scope of possible requirements to upper bounds in probabilities that a system may fail, rather than the number of faults.

ACKNOWLEDGMENT

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement n°269335 (MBAT) and the German Federal Ministry of Education and Research (BMBF) as well as $01IS12005M$ (SPES_XT).

 ISBN: 978-1-61208-401-5

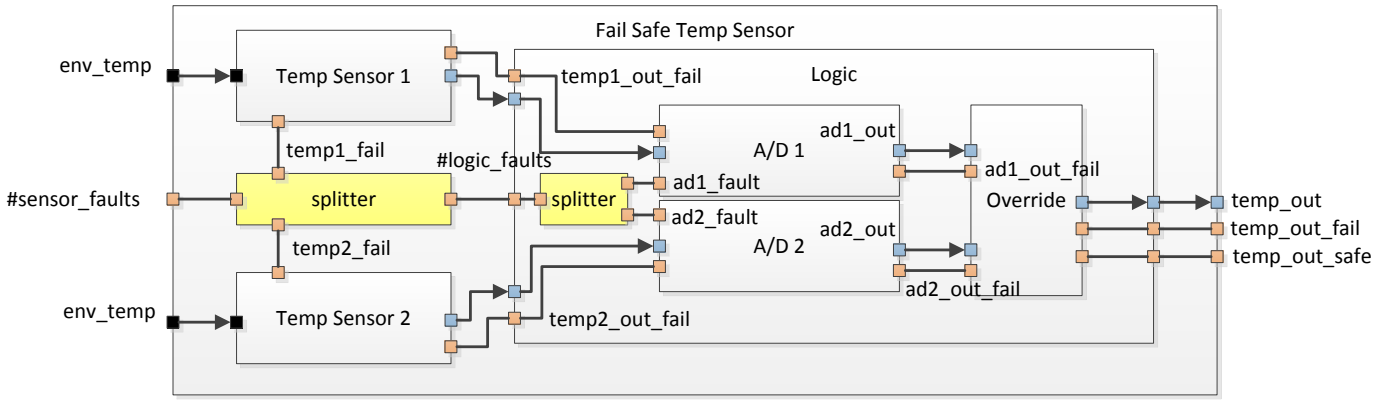

Figure 4. Architecture of a temperature sensor required to be robust against single-points of failure

REFERENCES

[1] J. L. Fenn, R. D. Hawkins, P. Williams, T. P. Kelly, M. G. Banner, and Y. Oakshott, "The who, where, how, why and when of modular and incremental certification," in System Safety, 2007 2nd Institution of Engineering and Technology International Conference on. IET, 2007, pp. 135–140.

[2] H. Espinoza, A. Ruiz, M. Sabetzadeh, and P. Panaroni, "Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems," in Software Certification (WoSoCER), 2011 First International Workshop on. IEEE, 2011, pp. 1–6.

[3] M. Oertel and A. Rettberg, "Reducing re-verification effort by requirement-based change management," in Embedded Systems: Design, Analysis and Verification. Springer Berlin Heidelberg, 2013.

[4] J. Fenn, R. Hawkins, P. Williams, and T. Kelly, "Safety case composition using contracts-refinements based on feedback from an industrial case study," in The Safety of Systems. Springer, 2007, pp. 133–146.

[5] J. Vara, S. Nair, E. Verhulst, J. Studzizba, P. Pepek, J. Lambourg, and M. Sabetzadeh, "Towards a model-based evolutionary chain of evidence for compliance with safety standards," in Computer Safety, Reliability, and Security, ser. Lecture Notes in Computer Science, F. Ortmeier and P. Daniel, Eds. Springer Berlin Heidelberg, 2012, vol. 7613.

[6] H. Hungar, "Compositionality with strong assumptions," in Nordic Workshop on Programming Theory. Mälardalen Real–Time Research Center, November 2011, pp. 11–13.

[7] H. Peng and S. Tahar, "A survey on compositional verification," Dep. of Electrical & Computer Engineering, Concordia University, Canada, Tech. Rep., 1998.

[8] M. Oertel, O. Kacimi, and E. Böde, "Proving compliance of implementation models to safety specifications," in Computer Safety, Reliability, and Security, ser. Lecture Notes in Computer Science, A. Bondavalli, A. Ceccarelli, and F. Ortmeier, Eds. Springer International Publishing, 2014, vol. 8696, pp. 97–107.

[9] Y. Papadopoulos and J. McDermid, "Hierarchically performed hazard origin and propagation studies," in Computer Safety, Reliability and Security, ser. Lecture Notes in Computer Science, M. Felici and K. Kanoun, Eds. Springer Berlin Heidelberg, 1999, vol. 1698, pp. 139–152.

[10] O. Lisagor, J. McDermid, and D. Pumfrey, "Towards a practicable process for automated safety analysis," in 24th International system safety conference, 2006, pp. 596–607.

[11] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from matlab-simulink models," in Dependable Systems and Networks, 2001. DSN 2001. International Conference on, 2001, pp. 77–82.

[12] J. Gould, M. Glossop, and A. Ioannides, "Review if hazard identification techniques," Health and Safety Laboratory, Tech. Rep., 2000.

[13] M. Wallace, "Modular architectural representation and analysis of fault propagation and transformation," in Proc. FESCA 2005, ENTCS 141(3), Elsevier, 2005, pp. 53–71.

[14] B. Kaiser, P. Liggesmeyer, and O. Mäckel, "A new component concept for fault trees," in Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software - Volume 33, ser. SCS '03. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 37–46.

[15] E. Böde, S. Gebhardt, and T. Peikenkamp, "Contract based assesment of safety critical systems," in Proceeding of the 7th European Systems Engineering Conference (EuSEC 2010), 2010.

[16] M. Oertel, A. Mahdi, E. Böde, and A. Rettberg, "Contract-based safety: Specification and application guidelines," in Proceedings of the 1st International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC 2014), 2014.

[17] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for systems design," Research Centre Rennes - Bretagne Atlantique, Tech. Rep., 2012.

[18] A. Baumgart, E. Böde, M. Büker, W. Damm, G. Ehmen, T. Gezgin, S. Henkler, H. Hungar, B. Josko, M. Oertel, T. Peikenkamp, P. Reinkemeier, I. Stierand, and R. Weber, "Architecture modeling," OFFIS, Tech. Rep., 03 2011.

[19] T. Henzinger, S. Qadeer, and S. Rajamani, "You assume, we guarantee: Methodology and case studies," in Computer Aided Verification, ser. Lecture Notes in Computer Science, A. Hu and M. Vardi, Eds. Springer Berlin Heidelberg, 1998, vol. 1427, pp. 440–451.

[20] W.-P. de Roever, "The need for compositional proof systems: A survey," in Compositionality: the significant difference. Springer, 1998, pp. 1–22.

[21] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in Formal Methods for Components and Objects, ser. Lecture Notes in Computer Science, F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Springer Berlin Heidelberg, 2008, vol. 5382, pp. 200–225.

[22] B. Delahaye, B. Caillaud, and A. Legay, "Probabilistic contracts: a compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects," Formal Methods in System Design, vol. 38, no. 1, 2011, pp. 1–32.

[23] A. Pnueli, "The temporal logic of programs," in Foundations of Computer Science, 1977., 18th Annual Symposium on. IEEE, 1977, pp. 46–57.

[24] K. Y. Rozier and M. Y. Vardi, "Ltl satisfiability checking," in Model Checking Software, Proceedings of the 14th International SPIN Workshop. Springer, 2007, pp. 149–167.

[25] J. Li, G. Pu, L. Zhang, M. Y. Vardi, and J. He, "Fast ltl satisfiability checking by sat solvers," arXiv preprint arXiv:1401.5677, 2014.

 ISBN: 978-1-61208-401-5