Validating Industrial Requirements with a Contract-Based Approach

Matthias Bernaerts*, Bentley James Oakes*, Ken Vanherpen*,
Bjorn Aelvoet†, Hans Vangheluwe*, and Joachim Denil*
*University of Antwerp and Flanders Make, Belgium
Matthias.Bernaerts@student.uantwerpen.be,
{Bentley.Oakes, Ken.Vanherpen, Hans.Vangheluwe, Joachim.Denil}@uantwerpen.be
†Dana Belgium NV, Belgium
Bjorn.Aelvoet@dana.com

***Abstract*—This paper presents our contract-based design technique for formalizing requirements during the design phase of a complicated and safety-critical automotive component. In our approach, contracts are created using property specification patterns to eliminate ambiguous unstructured natural language requirements, which could lead to misinterpretations or mismatched interfaces in the integration phases of the design process.**

These patterns are then automatically transformed into Signal Temporal Logic (STL) formulas. The STL formulas are verified on a modeled system of the component, utilizing the Matlab® toolbox Breach. This approach validates the industrial requirements described in the contracts, and can help achieve the requirement-based testing demanded by automotive safety standard ISO 26262.

***Index Terms*—Automotive, Requirement Validation, Formalizing Requirements, Contract-based Design, Property Specification Patterns, Signal Temporal Logic, Verification.**

I. Introduction

Cyber-Physical Systems (CPS) are distributed embedded computers that monitor and control physical processes; they assist us or take over tasks and are considered as one of the enablers of the fourth industrial revolution. CPS are becoming more complex and evolving at an enormous rate, presenting new challenges in all aspects of their design, such as the definition and validation of requirements and system specifications [1]. In particular, multi-disciplinary teams need to avoid inconsistencies to make the design process more efficient. This will reduce development costs of systems and the time to market [1]. Therefore, optimizing a design process is an important topic in industry.

A considerable part of this process is heuristic and vulnerable for misinterpretation, such as design techniques or requirements based on natural language. Formalization can eliminate the ambiguity and vagueness of requirements. This formalization assists industrial design processes for functional safety in adhering to the ISO 26262 standard [2], which concerns functional safety of electrical and/or electronic systems for road vehicles. The ISO 26262 standard places considerable emphasis on the verification of requirements with the purpose of identifying and eliminating inconsistencies and augmenting the overall quality of requirements.

The *Automated and Simulation-based Functional Safety Engineering Methodology* (aSET) project[1] is conducted by Flanders Make, the CoSys-Lab of the University of Antwerp and various industrial partners around the Flanders province of Belgium, such as DANA Belgium. The project's objective is to optimize design processes of functional safety components in the context of ISO 26262.

One effort of the project is introducing a contract-based approach to the design process of the industrial partners. In our work, textual informal requirements are (manually) written as formalized design contracts, which are then (automatically) translated into formal temporal logic. This formal logic has constructs which reason about the signals of the system, including timing constraints. An example is: *'when signal 1 turns positive, signal 2 becomes 100 within 2 sec.'*

The objective is to improve the quality of the requirements and avoid misinterpretations. Section IV-C demonstrates examples of where this formalization process was used to detect requirement misinterpretations. When requirements are transformed into formal logic, other improvements in the design process can also be accomplished. For example, the system can be validated by verifying whether the system signals satisfy the requirements defined by the formal logic. Advantages of this include a reduction in the time to write tests, as well as better traceability on the satisfaction of the requirements.

The core contributions of this paper are to explain the implementation of this contract-based approach on an industrial use-case, as well as provide examples of the benefits of this approach. Section II describes the industrial use-case, the current model-based design process, and a selection of requirements. Our contract-based design approach is then explained in Section III, along with a description of the contract language. Section IV details the specification of requirements in the contract language, the verification of the contracts on the system, and examples of inconsistencies encountered in the EDL requirements. Related work similar to our approach is presented in Section V, while we conclude with a pointer to future work in Section VI.

[1]https://www.flandersmake.be/en/projects/aset

II. Industrial Case Study

This section will present the case study used in this paper, which is a demonstrator model from an Electronic Differential Lock (EDL). This model has been designed by DANA Belgium, a company specialized in automotive driveline technologies.

An EDL is a system providing the functionality to enable and disable the wheels on the same axle to rotate at different speeds. The EDL can have an open differential, where the wheels are able to rotate at a different speed, or a closed differential where the wheels will travel at the same speed.

When the vehicle is turning in a curve, an open differential can prevent wheel slippage, as the wheel on the outside of the curve has to travel a greater distance and should therefore not be locked in speed with the other wheel. In contrast, a closed differential allows for more traction when accelerating from standstill on slippery terrain.

A. Design Process

Requirements for the EDL system were developed during the design process shown in Figure 1. This design process is a life-cycle development model based on the "V-cycle", which is common in the automotive industry [3]. This development model contains five different phases or life-cycles. Each phase has a decomposition and definition step on the left-hand side and a corresponding integration and verification step on the right-hand side.

First, the *stakeholder requirements* are negotiated and defined with a client. This contains a structural specification and the system's functional black box with interfaces. It also contains a behavioural specification consisting of the requirements allocated on the system with validation criteria, state machines, sequence diagrams, etc.

In the *functional concept* phase the black-box interfaces are refined and a functional breakdown of the system is performed. Requirements are also added and refined during this phase.

The next phase is the *system architecture* phase. In this phase the system is broken down in elements of different fields (electronics, mechanics, software, etc.) and their interfaces are defined. Requirements are allocated to the elements and are matched with validation criteria to be verified in the corresponding integration steps. In this design phase, more technical specifications are added.

In the fourth phase, the *software and hardware architecture* design phase, the elements of the previous phase are subdivided into smaller units. Requirements are again allocated to the smaller units and will be verified by validation criteria in the corresponding integration steps.

In the last design phase, the *component design* phase, the system is developed according to the requirements specified during the previous design phases.

Table I presents a selection of requirements from this EDL demonstrator model, along with the design phase in which they were defined. In Section IV, we will demonstrate how these requirements are transformed into verifiable contracts.

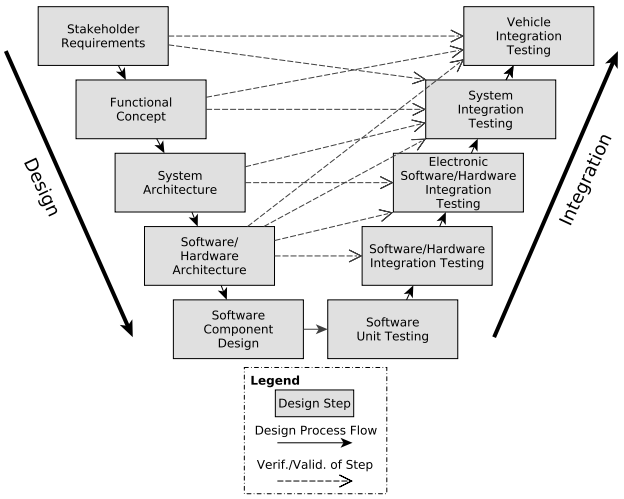

Fig. 1. The V-cycle design process.

B. Requirement Management

The design process followed in the case study, is a component-based design process, wherein complex systems are divided into smaller components. After the development of the different components, all the components are integrated to form the system. This design method might introduce some problems in the integration phase such as: incorrect interfaces, malfunctions, safety criteria that are not met, etc. The origin of those problems is often related to the management of the requirements. This because the traceability of requirements is lost between the different design phases.

Another reason can be that requirements are structured in different aspects or viewpoints, such as safety, mechanical, or software viewpoints. Often, these different viewpoints are developed by different people, teams or other companies all having different skills, frameworks and tools. This can lead to missing functionality or safety violations, which may not be caught until later in the development cycle increasing the cost to design and test the system.

In the existing design process, a model-based engineering approach is used. Meaning that for each phase, a system model is created or refined which contains the structure, behaviour, requirements and parametrics of the system [4].

In this paper, we address the mentioned issues in the current design process by introducing a contract-based approach.

III. Contract-based Design

Here the basis of contract-based design is presented, along with a description of how the design process in Section II-A is augmented with contracts. The contract language itself is then examined in the remainder of this section.

A. Contract-based Design Benefits

In *contract-based design* (CBD), a contract defines the assumptions of the environment and the guarantees of a component's behaviour under these assumptions [5]. This can enhance the design process of a system in the following ways:

Req. Name	Description	Originating Design Phase
Response to Driver Locking Command	*The system must close the EDL after receiving a locking command from the driver.*	Stakeholder Requirements
Inform Driver	*During normal operation and within* `t_SYSTEM_RESPONSE`*, the system must inform the driver about the EDL being in one of the following three operational states: EDL open state, or EDL close state, or EDL in transition between open and closed.*	Stakeholder Requirements
Verify EDL State	*Before reporting, the system must verify the EDL state by comparing the system EDL state with the physical position of the EDL and with vehicle dynamics data.*	Functional Concept
Logic Locking Command	*During normal operation (not during initialization or shutdown), the logic must generate a 12 Volt request to an analogue output hardware driver, within 50 milliseconds after receiving the rising edge of a pulse with valid duration. [This is only] if the current system EDL state is* `Open`*, and a falling edge pulse of minimal duration* `t_MIN_REQ` *and maximal duration* `t_MIN_REQ`*.*	System Architecture

TABLE I
SELECTED REQUIREMENTS FOR THE ELECTRONIC DIFFERENTIAL LOCK (EDL) CASE STUDY.

- Addressing of integration complexity. Contracts can define an architectural description that outlines the interfaces between the components, the used units, value ranges, etc. This avoids inconsistencies between different subsystems and problems in the integration phase.
- Better management of requirements. This because requirements can only be tested on implementations. Likewise, requirements cannot be simulated in most cases. Contracts offer improved support for proving the satisfaction of requirements.
- Addressing of complex chains between *original equipment manufacturers* (OEMs) and suppliers by decomposing contracts of the needed subsystems.

B. Contract-Based Design Process

The contract-based design can be integrated into the current model-based design process. Figure 2 shows the new design process where the contract-based approach is implemented. Note that Figure 2 focuses on the first four stages of the V-cycle in Figure 1: *Stakeholder Requirements*, *Functional Concept*, *System Architecture*, and *Software/Hardware Architecture*.

In the new process, requirements are specified as contracts in the different life-cycles of the systems development. These contracts enable early validation of the system, improving the design process. For example, models are created during the *Functional Concept* stage which define the system functionality. These models can then be used to simulate the system, and validate the requirements (described as contracts) from the *Stakeholders Requirements* and *Functional Concept* stages.

This early validation helps designers to determine if the functionality of a system is properly defined during the *Functional Concept* phase, before determining the physical implementation. During the following *System Architecture* and the *Software/Hardware Architecture* design phases, additional contracts will be specified. During the component design, testing, and integration phases of the system's development, this set of contracts provides a structured process to validate requirements and improve the verification process of a system.

C. Contract Language

This section and the following will briefly describe the constructs contained in the contract language, such as the

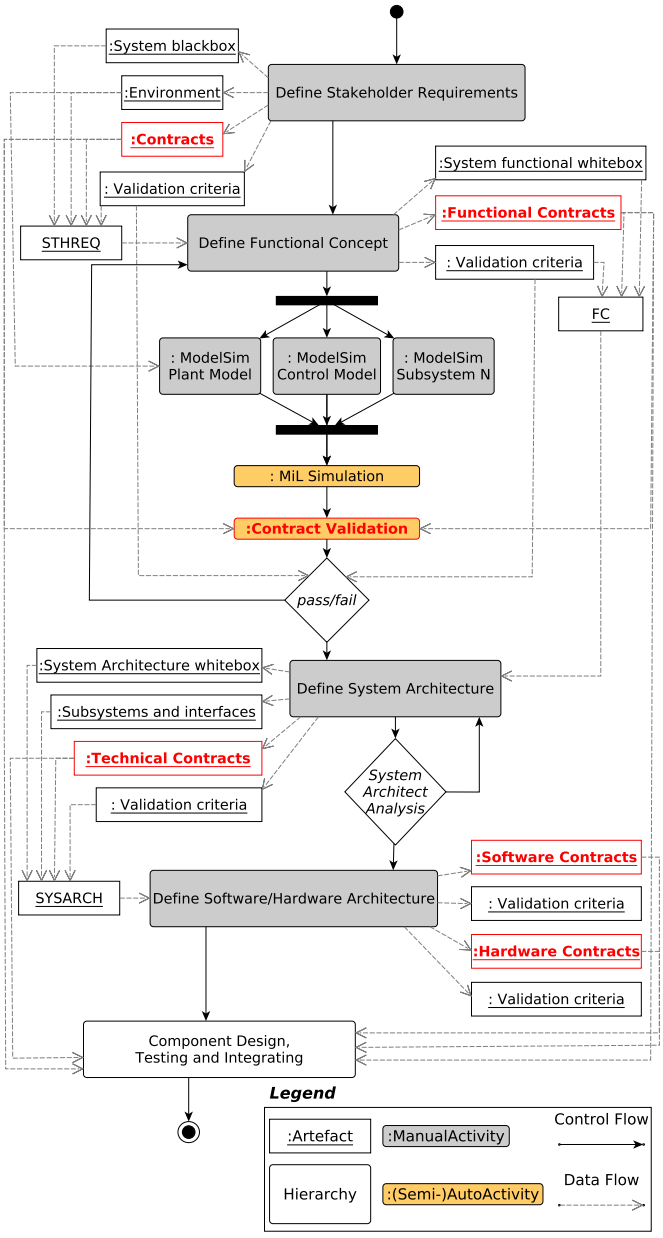

Fig. 2. Process model for contract-based design approach.

```
Contract ReqID{
    longname "Requirement Name"
    description "Requirement Description"

    statements{
        Event event1 := ToBeDetermined,
        Property property1 := ToBeDetermined
    }
    scope Globally
    pattern Universality:
        always-the-case-that event1 holds
}
```

Fig. 3. Basic format of a contract which contains a placeholder description, event, property, scope, and pattern.

temporal logic constructs which are encoded in property specification *scopes* and *patterns*. As well, the operators used to specify *statements* on system signals will also be presented. The contract language is still under development in the aSET project, and the full formalization of the syntax and semantics will be presented in future work.

Figure 3 presents the basic format of a contract. The first line provides a short ID for the contract, while the `longname` field provides a human-readable version of the name. The `description` stores the informal textual requirement, which is important to retain for comprehension of the contract.

As well, as the statements, scope, and pattern of the contract are optional, the first step in the development of a contract may be to only have the textual description. Note however that such a contract would not be able to be verified, as discussed in Section IV-D.

In our approach, the requirement engineer must manually interpret the natural language text to create the appropriate contract, as automatic contract production is out of the scope of our approach. This manual process can lead to misinterpretations, as shown in Section IV-C. However, the next step of transforming the contract language into a formalized logic is performed automatically (see Section III-F1), to abstract the complex temporal logic formulas away from the requirement engineer.

Linking the informal natural language of the requirement description with a formalized contract language and the subsequently generated formal logic assists in making the requirement less ambiguous. Section IV-C discusses how this formalization improves requirement quality as the resulting temporal logic has a precisely defined syntax and semantics.

D. Temporal Logic

System requirements, especially safety-critical requirements, relate to the notion of time. Some example requirements are:

- When a button is pushed, *eventually* an action will be performed;
- *Before* this error signal is sent, this light is *always* off;
- *After* passing these environmental thresholds, the system will *never* be guaranteed to work again;

```
alw((DWS[t] == 1 and vehicleEDLState[t] == EDL_OPEN)
or
(((Cornering[t] == 1 and DWS[t] == 0) and vehicleEDL-
State[t] == EDL_CLOSED) or
((DrivingSlow[t] == 1 or (Cornering[t] == 0 and DWS[t]
== 0)) and vehicleEDLState[t] == EDL_DK)))
```

Fig. 4. The STL formula in Breach syntax automatically generated for the TR57 contract.

- A system will be powered *until* a certain condition is met.

Temporal logic enables reasoning about time in a formal way, and allows for verification given a set of system signals as in Section IV-D.

There are different types of logic existing under the category of temporal logic, such as Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL). This contract language targets Signal Temporal Logic (STL), which is an extension of MTL which deals with continuous signals in a system as in the systems under study. In particular, STL contains predicates over real values, which enable the contract language to reason about signal values as events [6].

E. Property Specification Patterns

Reasoning about and writing correct temporal logic is a time-consuming and challenging task that often requires an expert [7]. As an example of this, Figure 4 shows the STL formula which has been automatically generated for the TR57 contract in Figure 10. The formula is dense and difficult to reason about yet it only contains one temporal operator (*always*). Note that this contract is verified by the Breach toolbox in Figure 13.

To overcome the problem of manually specifying temporal patterns, Dwyer et al. [8] developed *property specification patterns*. Each pattern in this collection maps directly to an STL formula, and yet they retain a more natural language format. Thus, property specification patterns are a pragmatic way to write STL formulas while maintaining a familiar syntax for the requirements engineer.

The patterns in the contract language for the aSET project are based on the pattern catalogue of Autili *et al.* [7]. This improvement over Dwyer refers to *events* in the system, and how they relate to each other in time. The pattern catalogue includes:

- **Absence**: P may never occur.
- **Universality**: P is always true. Negation of the absence pattern.
- **Existence**: P occurs at least once inside the scope.
- **Bounded Existence**: an extension of the *Existence* pattern, also defining how many times P occurs.
- **Steady State**: P will eventually hold.
- **Transient State**: P will hold after exactly t time units.
- **Recurrence**: P has to occur at least once during each t time units.
- **Min Duration**: whenever P occurs it must hold at least t time units.

- **Max Duration**: whenever P occurs it must hold less than t time units.
- **Precedence**: if P holds, then it must have been the case that S has occurred.
- **Until**: P holds without interruption until S holds.
- **Response**: if P has occurred, then in response S holds continually.
- **PrecedenceChain1N**: if S has occurred and afterwards multiple events (Q, R, ...) hold, then it must have been the case that P has occurred.
- **PrecedenceChainN1**: if P holds, then it must have been the case that S and afterwards multiple events (Q, R, ...) have occurred before P holds.
- **Response1N**: if P has occurred, then in response S eventually holds, followed by multiple events (Q, R, ...).
- **ResponseN1**: if S followed by multiple events (Q, R, ...) have occurred, then in response P eventually holds.

Most of these patterns can also be augmented with timing constraints. For example, the *ResponsePattern* pattern can include the condition that the event S occurs within 50 milliseconds of P occurring. This is shown in a contract example in Figure 10. This increase in expressiveness is required for the safety-critical requirements studied in this research.

These patterns are then augmented with *scopes*, which define when the pattern must hold. Section IV contains multiple examples of their interaction. Scopes are useful in a complex system, as these systems have a notion of states where the behaviour of the system changes.

The five scopes available in the contract language are:

- **globally**: the pattern always holds.
- **before** Q: the pattern holds before Q.
- **after** Q: the pattern holds after Q.
- **between** Q **and** R: the pattern holds between Q and R.
- **after** Q **until** R: the pattern holds after Q until R.

F. Statements

In the patterns and scopes, the statements P, Q, R and S are events, which must relate to the signals of the system being verified. However, the property specification pattern literature does not provide operators for the cyber-physical systems which we study. Thus, one component of the development of the contract language is a sub-language for defining *statements*. This sub-language can refer to signals in the system, and offers mathematical and engineering operators for reasoning about those signals.

As an example of statements, a requirement from the EDL is: *"the system must close the EDL after receiving a locking command from the driver"*. This requirement needs two statements to be described in the contract language which are shown in Figure 5. The first is an *event*, which represents occasional conditions occurring throughout the system's operation. The event in the requirement `driver_lock` concerns the receiving of the locking command. The definition of this event is that the system signal `DRIVER_LOCK`, which is a Boolean value, becomes true.

```
statements{
    Event driver_lock := DRIVER_LOCK == True,
    Property close_EDL := EDL_STATE in Set { EDL_State CLOSED }
}
```

Fig. 5. Example of an event and a property for a contract.

In contrast, a *property* in our contract language describes system statements that hold for some amount of time after becoming true. Note that this is closely related to the notion of system *state*, though it is slightly broader as it can include transient states. Figure 5 shows the `close_EDL` property, which detects when the EDL is closed. This is defined by checking whether the system signal `EDL_STATE` is in the state `EDL_State CLOSED`.

A statement must be true or false to be used in the patterns and scopes. Therefore the definition of these statements is a predicate over the states and signals in the system, involving operators such as comparisons. These predicates are evaluated at each point of time in the system by the underlying temporal logic checker. In this approach, we employ the Breach toolbox as discussed in Section IV-D.

1) Contract Language Editor: The development of the contract language has also involved the creation of an editor for the requirement engineer to easily specify these contracts. The framework for defining the contract language and editor is XText[2]. Benefits of this platform include syntax highlighting, auto-completion, template support, and error-checking.

As well, contracts are transformed into Signal Temporal Logic (STL) formulas to be verified. XText allows the automatic generation of those formulas whenever the contract file is saved, reducing the burden on the requirement engineer to produce those files. This transformation occurs whenever the keyword `generate-STL` is written at the bottom of a contract.

IV. From Requirements to Validated Contracts

This section will highlight the process in which a requirement is transformed into a contract and then verified. As well, concrete examples of improving requirement quality will be highlighted.

A. System Models

The context for employing contracts is during the design of a safety-critical component. To support this activity, the contract language can refer to elements of a system design model.

As part of the aSET project, a language definition for designing safety-critical systems has been developed. This language definition is termed a *meta-model*, which contains the types, structure, rules and constraints available for writing another model [9]. This meta-model, named *Formal Functional Safety Model* (FuSaFoMo), supports the functional safety artefacts of the ISO 26262 standard, like *elements*, *items*, etc. This FuSaFoMo enables an architectural description of the system.

[2]https://www.eclipse.org/Xtext/

This is a similar description to that given by a block definition diagram, internal block diagram, and activity diagram in SysML [4]. For this industrial case study, an instance of the FuSaFoMo meta-model for the Electronic Differential Lock (EDL) was created, defining the design of the component. Additionally, a Simulink model was (manually) produced as an implementation of the behaviour to be used in the verification checks (see Section IV-D).

Once a system model that is an instance of the Formal Functional Safety model is created, the contract language can then refer to ports, connections, parameters, and states from the EDL model. These system elements are used as the source of signals to be used in the definition of properties and events.

Note that if a system model is not available, signal names can still be written in the contracts, and the link to the system model performed later. This decoupling was requested by our industrial partners as a way of offering flexibility in the contract development process. The requirements engineer should not be blocked on the definition of the system model, as this could be performed by a separate team. Textual references can be used to design the contract, although the requirement engineer must manually check that the signal names match the simulated model during the verification process (see Section IV-D).

```
Contract FR07{
        longname "Response to driver locking command"
        description "The system must close the EDL after
                     receiving a locking command from the driver."

        statements{
            Event driver_lock :=
                Port driverCommands_closingRequest == True,
            Property close_EDL :=
                EDL_STATE in Set{State EDLphysicalSyst_CloseEDL}
        }
        scope Globally
        pattern ResponsePattern:
            if driver_lock has-occurred,then-in-response
                close_EDL eventually-occurs
        generate-STL
    }
```

Fig. 6. Contract FR07, closing the EDL in response to a driver command.

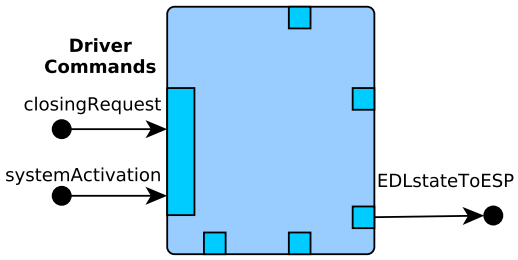

Fig. 7. The EDL system as a black box component with incoming and outgoing ports.

B. Defining Contracts

An example process of defining a contract and its connection to the EDL system model will be presented in this section. The requirement selected is: *"the system must close the EDL after receiving a locking command from the driver"*, which will be written as the contract in Figure 6.

1) Identification: The contract `ID`, `longname`, and `description` are created from this textual requirement. Note that most likely the requirement is directly copied into the description.

Then, either the statements for the contract or scope and pattern will be defined. This explanation will focus on the statements first, but for some contracts it may be preferable to reason about the scope and pattern first.

2) Statements: As indicated in Table I, this requirement is from the *Stakeholder Requirements* design phase of the EDL component. Therefore, only a black box view of the component is available, shown in Figure 7. This view hides the internal details of the component and only exposes the outside port connections.

The event `driver_lock` in the contract will refer to a port in this black box model. The keyword `Port` is used to indicate that this event refers to the port `driverCommands_closingRequest`. Note that the naming scheme indicates that the port `closingRequest` is part of the interface `driverCommands`.

The property `close_EDL` must obtain the state of the EDL from the black box model. However, as the state is not directly exposed by the black box model in a port, no explicit connection can be made at this stage. This connection will be made at a later point in the design process when more information is available.

3) Scope and Pattern: Either before or after the contract statements are defined, the scope and pattern of the requirement can be created. As the requirement states that the system must perform an action in response to receiving a command, the *Response Pattern* is most appropriate. The command becomes the first statement in the pattern, and the state checking is the second statement.

The scope of a contract is chosen by examining when the pattern is valid for the system. As the enabling condition are not specified, the assumption could be made that this requirement is always valid. In this case, a *Global* scope would be chosen.

However, this assumption may not be valid. If the system is unpowered or in an error state, then this requirement does not hold. This imprecision arises because this requirement was defined early on in the EDL design process, in the *Stakeholder Requirement* phase. In later phases, this requirement will be refined to become more precise. Nevertheless, the contract language assists the requirements engineer in detecting this incompleteness, by providing the constructs to reason about which periods of system state enable a pattern.

C. Improving Requirement Quality

The integration of the contract language in the design process for the EDL component uncovered a number of issues with requirement quality. Here two cases are presented to illustrate how the quality of the requirements can be improved by using the contract language, which has a positive impact on our industrial partners.

```
Contract FR12{
        longname "Report to ESP"
        description " During normal operation and within
        t_SYSTEM_RESPONSE, the system must report to the ESP that
         the EDL is in one of the following two states:
        - Open; or
        - Closed.

        Unless the EDL is confirmed open, the system must report
        to the ESP that the EDL is closed."

        statements{
            Event enter_error_state := EnterErrorState == True,
            Event ReportState := EDLStateToESP from-table
                case SYSTEM_EDL_State != VERIFIED_EDL_State
                    : Set {EDL_State CLOSED}
                case SYSTEM_EDL_State == VERIFIED_EDL_State
                    : Set{ EDL_State OPENED, EDL_State CLOSED }
        }
        scope Before enter_error_state
        pattern Recurrence:
            ReportState occurs-repeatedly every 10 ms
    }
```

Fig. 8. Contract FR12, periodically reporting the state of the EDL.

1) Ambiguous Requirement: The use of natural language for specifying requirements can lead to ambiguity. For example, the FR12 requirement states: *"During normal operation and within `t_SYSTEM_RESPONSE`, the system must report to the ESP that the EDL is in one of the following two states: Open or Closed. Unless the EDL is confirmed open, the system must report to the ESP that the EDL is closed"*.

One interpretation of this requirement is that when the system is queried, it must respond with the state of the EDL, within the duration `t_SYSTEM_RESPONSE`. This contract would therefore have a *Global* scope and a *ReponsePattern*.

However, upon formalization of this contract and subsequent validation with the requirement engineer, it was discovered that this is the wrong interpretation. In fact, `t_SYSTEM_RESPONSE` is the period of time between when the EDL state is communicated to the Electronic Stability Program (ESP). Clearly these are very different semantics, which must be defined as early as possible in the EDL design process.

Figure 8 shows the correct version of this contract. The pattern is a *Recurrence Pattern* where the `ReportState` event occurs periodically. Note that in this version of the contract, the period `t_SYSTEM_RESPONSE` has been refined to be 10 milliseconds.

2) Requirement Conflict: As the number of requirements for a system grows, it becomes more likely that there will be conflicts in how the requirements state that the system should behave. For example, the EDL case contains two requirements FR14 and FR10. FR14 states the following: *"The EDL must remain physically closed for a duration `t_POSTPONE_UNLOCK` after a system shutdown request"*. However, FR10 states the following: *"If the EDL is not yet open at system initialization, it must be opened"*. There is a conflict in that if system shutdown and system initialization occur within the duration `t_POSTPONE_UNLOCK`, it is ambiguous whether the EDL should be opened or not.

In fact, this ambiguity was resolved in the metadata for the requirements, which stated that the initialization behaviour had a priority over the shutdown behaviour. In the contract language, this information is best formalized in the contract itself to avoid misinterpretation. The rewritten FR14 contract is presented in Figure 9. The description of the contract has been updated, and the scope of the contract specifies that the pattern is only valid after the shutdown event until the initialization event is received.

```
Contract FR14{
        longname "Shutdown behavior when EDL is closed"
        description "If the EDL is closed upon receiving
        a shutdown request, the system has to keep the
        EDL closed until any of the following conditions
        is satisfied:
        - a time t_POSTPONE_UNLOCK has elapsed; or
        - an initialization request is received."

        statements{
            Event system_shutdown_when_closed :=
                AND {
                    systemActivation == False,
                    STATE in Set{EDL_State CLOSED}
                },
            Event system_init :=
                SystemInit == True,
            Property edl_closed :=
                STATE in Set{EDL_State CLOSED}
        }
        scope After system_shutdown_when_closed
              Until system_init
        pattern MinDuration:
            once edl_closed becomes-satisfied,
                it-remains-so-for-at-least 10 ms
        generate-STL
    }
```

Fig. 9. Contract FR14, specifying the minimum duration of system shutdown.

D. Verification of STL

The contracts in the contract language are automatically mapped directly to Signal Temporal Logic (STL), as stated in Section III-E. This temporal logic is then verified on the simulation traces of the system, with a negative result indicating that the represented requirement is not satisfied by the system. However, as this approach is simulation-based, if the temporal logic is satisfied, this does not guarantee that the system satisfies the requirement in all cases. This approach to verification therefore requires that appropriate test scenarios are provided for full coverage of the system.

As well, other system effects may also need to be taken into account, depending on the kind of requirement and the test performed. For example, in Table I on page 3, the requirement *logic locking command* includes a notion of physical delays, such as the worst-case executing time for a system software component. These delays need to be reproduced in any model-in-the-loop simulation, to simulate any software or hardware delays present in the system.

Despite the above limitations, this contract-based verification approach for validating requirements allows improvements to be made throughout the design process, such as early detection of errors in the early development phases, or for validation of requirements throughout. These contracts can also be used throughout the testing phase of the component

```
Contract TR57{
        longname "Logic vehicle dynamics EDL state"
        description "within 50ms after receiving data
            from the CAN bus, the logic must determine
            the EDL state from vehicle dynamics:
            • Vehicle EDL state is 'OPEN' if a differential
            wheel speed is registered;
            • Vehicle EDL state is 'CLOSED' if
            no differential wheel speed is registered
            while the vehicle is cornering;
            • Vehicle EDL state is 'UNKNOWN' if the vehicle
            is driving slow or there is no differential wheel
            speed while the vehicle is not cornering."
        statements{
            Event receiveDataCAN := newCANdata == True,
            Event vehicle_EDL_State :=
                EDLVehicleState from-table
                    case DWS==True
                        : Set{EDL_State OPENED}
                    case and{ DWS==False, Cornering==True}
                        : Set{EDL_State CLOSED}
                    case DrivingSlow==True
                        : Set{EDL_State UNKNOWN}
                    case and{ DWS==False, Cornering==False}
                        : Set{EDL_State UNKNOWN}
        }
        scope Globally
        pattern ResponsePattern:
            if receiveDataCAN has-occurred,
                then-in-response vehicle_EDL_State
                eventually-occurs within 50ms
        generate-STL
    }
```

Fig. 10. Contract TR57, which determines the EDL state according to vehicle dynamics.

design process, reducing the number of other assessments that need to be developed.

1) Breach Toolbox: In this paper the Matlab toolbox Breach [10] is used to verify the STL formulas. In particular, this verification is performed on simulations of Simulink system architecture models, which are (manually) created by engineers to fully implement the behaviour of the system. Note that currently there is no check that the Simulink model conforms to the specification detailed in the Functional Safety Formal Model meta-model, described in Section IV-A.

The Breach toolbox receives as input the STL contracts, and the Simulink model of the system. As output, the Boolean satisfaction of each STL formula is reported, as well as the *quantitative satisfaction*. This quantitative satisfaction (also termed *robustness*) demonstrates how far the STL formula is from being true or false [11]. This information is useful in a safety-critical context, as a requirement which almost fails also requires attention.

2) Example: As an example, the TR57 requirement defines part of the safety logic that determines in which state the EDL is, according to the vehicle dynamics. In Figure 10, the contract of this requirement is shown, including the textual description. The scope is *Global*, as this contract should hold at anytime. The pattern is a *ResponsePattern*, as when the `receiveDataCAN` event is received, the system must determine the EDL state, represented by the `vehicle_EDL_state` event.

The first statement in this contract, `receiveDataCAN`, refers to when data is received from the CAN data bus in the vehicle, containing data from the vehicle sensors such as wheel

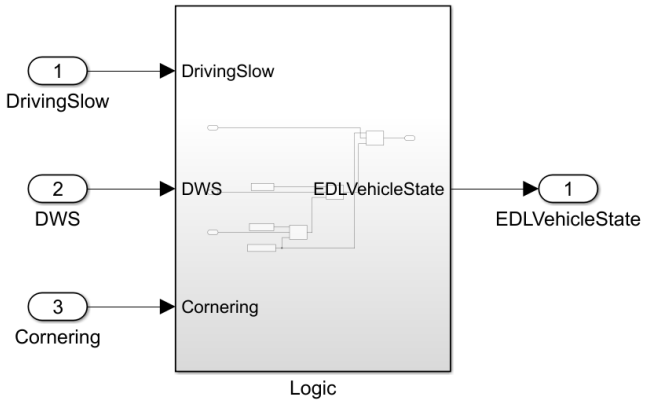

Fig. 11. Simulink component to determine EDL state of the vehicle

```
scope Globally
pattern Universality:
    always-the-case-that vehicle_EDL_State holds
```

Fig. 12. Version of TR57 used for verification in Breach.

speeds. The second statement, `vehicle_EDL_state`, defines the logic that will determine the state of the EDL depending on whether the the vehicle is driving slow, cornering or has a differential wheel speed (DWS).

This logic is implemented as a table which determines the desired value of `EDLVehicleState` depending on different cases. For example, if the `DWS` signal is false and the `Cornering` signal is true, then the EDL state must be closed. Note that this statement assumes that `DWS`, `Cornering` and `DrivingSlow` are signals in the system, but this has not been linked to an explicit system model as explained in Section IV-A. Instead, these names come from the implementation of this component in Simulink, shown in Figure 11.

For the verification activity discussed next, this contract and component were tested in isolation. As the CAN signal was not represented in this model, a version of the TR57 contract was created with only the `vehicle_EDL_State` event, a *Global* scope, and the *Universality* pattern where the event should always hold. This version is shown in Figure 12.

3) Verification in Breach: Verification of STL formulas is performed in the following steps: a) the Simulink model under study is loaded, b) the Breach toolbox is initialized within Matlab, c) the model is parameterized and simulated to log the signals in the Matlab toolbox, d) the STL formulas are loaded by Breach and verified against the signals, e) the Boolean and quantitative satisfaction for each STL formula is presented to the user, along with the decomposition of each formula.

The satisfaction plots for the TR57 contract are shown in Figure 13. The satisfaction for the complete STL formula is found in the bottom graph, with the textual formula above. The three upper graphs are sub-formulas verified throughout the entire simulation time, allowing the verification engineer to understand the satisfaction result for the full formula.

For each graph, the red line demonstrates the Boolean

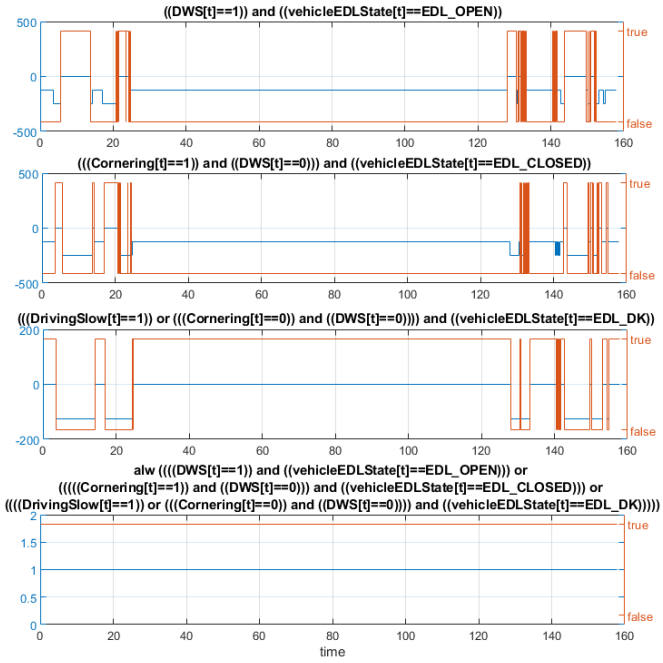

Fig. 13. Contract TR57, Boolean and quantitative satisfaction in Breach.

satisfaction for the formula throughout time, while the blue line shows the quantitative satisfaction. This robustness information allows the verification engineer to examine how close or far the formula was from failing. Note that for the complete formula, the Boolean satisfaction is true throughout the simulation. Therefore, this contract is satisfied.

V. Related Work

Past literature has focused on improving and optimizing design processes for industry, in an effort to lower time used, cost, and number of errors.

A. Requirements

In the field of requirement engineering, ambiguity, complexity and vagueness are common problems. The Easy Approach to Requirements Syntax (EARS) [12] is one method to solve those problems. In this method, a ruleset with five simple templates is introduced. By using the EARS templates, requirements in an unstructured natural language are forced into a simple structure, which reduces the complexity for many requirements. Note that in our work, our industrial requirements must be more specific, so we used the property specification patterns combined with a statement language to refer to signals.

The work of Meyers [13] builds contracts in a different fashion to us. In that work, a domain-specific language is built such that the engineer can refer to components in the system at a high level of abstraction. This domain-specific language is then transformed into a graphical language where the user can build temporal logic formulas in the Linear Temporal Logic (LTL) formalism. For example, the user can specify that for all floors, after an elevator call button is pressed, the elevator should eventually arrive. In contrast, our contract language is built on STL which enables us to reason about signals.

B. Modeling of Functional Safety

The Functional Safety Formal Model (FuSaFoMo) described in Section IV-A is similar to other formalizations of the functional safety domain found in the literature. For example, Taguchi provides meta-models for Failure Mode and Effects Analysis (FMEA) and ISO 26262, as well as their alignment [14]. Extraction of meta-models from safety standards is presented as an approach by Luo *et al.* [15], and a generic meta-model for "reference assurance frameworks" has been created by de la Vara *et al.* [16]. In contrast to these approaches, the FuSaFoMo focuses on the concept phase of ISO 26262, such as item definition, hazard analysis and risk assessment, and safety-critical functions.

C. Contracts and Contract Verification

Contract-Based Co-Design (CBCD) [1] is an approach focusing on ensuring consistency of multiple viewpoints in the design process. This allows the translation of contracts to viewpoint-specific contracts, such as view for an electrical engineer and another for a thermal engineer. As a result, the inconsistencies arising from operating in different viewpoints on a requirement can be reduced.

This paper performs verification of STL contracts on traces of a simulation. Another approach is to apply assume-guarantee reasoning on architectural models. Examples of this include one of the first tools to verify contract refinement for embedded systems, the Othello Contracts Refinement Analysis (OCRA) tool [17], or the Assume Guarantee Reasoning Environment AGREE tool [18]. Both of these tools verify Linear Temporal Logic (LTL) constraints which are defined on an architectural model similar to the Functional Safety Formal Model described in Section IV-A.

The Breach tool is used in our paper to verify our STL formulas and calculate the level of satisfaction of a formula on a signal (termed *robustness*) [19]. Another similar tool, which operates on an extension of STL, is AMT 2.0 [20]. AMT also allows trace diagnostics on top of property satisfaction checking, by using the error diagnostics algorithm described in [21]. Balsini *et al.* [22] also introduced a tool to generate Simulink monitors from STL constraints, enabling designers to analyze system specifications during Simulink simulations.

For verification at system runtime, the contract monitor must run directly on the deployed system. It is important that the verification does not interfere with the analyzed system. A tool that accomplishes this is *Brace*, a runtime verification system that allows efficient, and scalable runtime monitoring [23].

The Simulink Oracles for CPS Require-menTs with uncErtainty (SOCRaTes) approach [24] defines the Signal First Order logic (SFOL) language for defining requirements for cyber-physical systems, which subsumes STL. SFOL is then transformed into Simulink monitor blocks, which provide robustness information and can terminate simulations if errors are detected. Although the approach is very similar, our

approach provides a contract language for the requirement engineer which is closer to natural language with the definition of scopes and patterns.

VI. Conclusion

Improving the design process of a system can reduce the development cost and time to market. In this paper, we focus on the formalization and partial verification of requirements to remove ambiguity and detect issues early. This formalization is achieved with the definition of a contract language and the automatic mapping of those contracts into temporal logic which can be verified on a simulation.

The ISO 26262 standard [2] demands requirement-based testing. Validation of requirements is necessary to achieve process compliance, especially for systems with higher *Automotive Safety Integrity Levels* (ASIL). These ASIL are mandated by ISO 26262 to require greater risk management, along with increased traceability during the design process. This paper shows that an early validation of requirements is possible during model-in-the-loop (MiL) testing, helping achieve compliance of the design process with the ISO 26262 standard.

This paper has presented brief details about the contract language, such as the property specification patterns and the statement language. As well, we show industrial requirements from our case study that have been found to be ambiguous by being formalized in the contract language. This formalization therefore results in the reduction of misinterpretations and improvement of the quality of the requirements through during the different stages of the design process.

As an industrial partner in the aSET project, Dana Belgium NV has seen great potential in using the contract-based requirements technology in the design and development of safety-critical functions. "Safety functions" are especially suited for this as they need to be simple (as in non-complex) in nature, and formalization helps in attaining the (process) requirements for the higher safety integrity levels. Dana is planning on incorporating the contract-based requirements technology as part of the current day-to-day practices for validating safety-critical systems.

For future work, we plan to validate further industrial requirements. This involves collaboration with our industrial partners to improve and extend the concepts, statements, and validation methods for the contract language, and improve the usability of our approach. Additionally, the full syntax, semantics, and expressiveness of the contract language must be precisely defined to allow the requirement engineer to reason about which system properties can be expressed and verified.

Acknowledgment

The authors thank the anonymous reviewers for their feedback and suggestions. This work was partly funded by Flanders Make vzw, the strategic research centre for the Flemish manufacturing industry; and by the Flanders Make project aSET (grant no. HBC.2017.0389) of the Flanders Innovation and Entrepreneurship agency (VLAIO).

References

[1] K. Vanherpen, "A contract-based approach for multi-viewpoint consistency in the concurrent design of cyber-physical systems," Ph.D. dissertation, University of Antwerp, 2018.

[2] International Organization for Standardization, "ISO 26262: Road vehicles-functional safety," 2011.

[3] J. A. Estefan *et al.*, "Survey of model-based systems engineering methodologies," *Incose MBSE Focus Group*, vol. 25, no. 8, pp. 1–12, 2007.

[4] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[5] A. Benveniste *et al.*, "Contracts for systems design: Theory," Ph.D. dissertation, Inria Rennes Bretagne Atlantique; INRIA, 2015.

[6] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.

[7] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*. IEEE, 1999, pp. 411–420.

[9] T. Kühne, "Matters of (meta-) modeling," *Software and Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006.

[10] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 167–170. [Online]. Available: https://github.com/decyphir/breach

[11] A. Donzé, T. Ferrere, and O. Maler, "Efficient robust monitoring for STL," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 264–279.

[12] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *2009 17th IEEE International Requirements Engineering Conference*. IEEE, 2009, pp. 317–322.

[13] B. Meyers *et al.*, "Promobox: A framework for generating domain-specific property languages," in *International Conference on Software Language Engineering*. Springer, 2014, pp. 1–20.

[14] K. Taguchi, "Meta modeling approach to safety standard for consumer devices," in *Seminar on Systems Assurance & Safety for Consumer Devices*, 2011.

[15] Y. Luo *et al.*, "Extracting models from ISO 26262 for reusable safety assurance," in *International conference on software reuse*. Springer, 2013, pp. 192–207.

[16] J. L. de la Vara *et al.*, "Model-based specification of safety compliance needs for critical systems: A holistic generic metamodel," *Information and software technology*, vol. 72, pp. 16–30, 2016.

[17] A. Cimatti *et al.*, "Ocra: A tool for checking the refinement of temporal contracts," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 702–705.

[18] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha, "Compositional verification of architectural models," in *NASA Formal Methods Symposium*. Springer, 2012, pp. 126–140.

[19] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2010, pp. 92–106.

[20] D. Ničković, O. Lebeltel, O. Maler, T. Ferrère, and D. Ulus, "Amt 2.0: qualitative and quantitative trace analysis with extended signal temporal logic," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 303–319.

[21] T. Ferrère, O. Maler, and D. Ničković, "Trace diagnostics using temporal implicants," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2015, pp. 241–258.

[22] A. Balsini *et al.*, "Generation of Simulink monitors for control applications from formal requirements," in *IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2017, pp. 1–9.

[23] X. Zheng, C. Julien, R. Podorozhny, F. Cassez, and T. Rakotoarivelo, "Efficient and scalable runtime monitoring for cyber–physical system," *IEEE Systems Journal*, vol. 12, no. 2, pp. 1667–1678, 2016.

[24] C. Menghi, S. Nejati, K. Gaaloul, and L. Briand, "Generating automated and online test oracles for simulink models with continuous and uncertain behaviors," *arXiv preprint arXiv:1903.03399*, 2019.