Towards Development of Complete and Conflict-Free Requirements

Niskayuna, NY

Abha Moitra, Kit Siu,
Andrew W. Crapo,
Harsh Chamarthi, Michael Durling,
Meng Li, Han Yu
GE Global Research

Panagiotis Manolios
Northeastern University
Boston, MA

Michael Meiners
GE Aviation Systems
Grand Rapids, MI

Abstract—**Writing requirements is no easy task. Common problems include ambiguity in statements, specifications at the wrong level of abstraction, statements with inconsistent references to types, conflicting requirements, and incomplete requirements. These pitfalls lead to errors being introduced early in the design process. The longer the gap between error introduction and error discovery, the higher the cost associated with the error. To address the growing cost of system development, we introduce a tool called ASSERT™ (Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests) for capturing requirements, backed by a formal requirements analysis engine. ASSERT™ also automatically generates a complete set of requirements-based test cases. Capturing requirements in an unambiguous way and then formally analyzing them with an automated theorem prover eliminates errors as soon as requirements are written. It also addresses the historical problem that analysis engines are hard to use for someone without formal methods expertise and analysis results are often difficult for the end-user to understand and make actionable. ASSERT™'s major contribution is to bring powerful requirements capture and analysis capability to the domain of the end-user. We provide explainable and automated formal analysis, something we found important for a tool's adoptability in industry.**

Index Terms—**Requirements Formalization, Formal Analysis of Requirements, Ontology, Automated Requirements-Based Test Generation, Requirements Engineering, Formal Methods**

I. Introduction

The General Electric (GE) Company designs, develops, verifies and deploys software intensive critical infrastructure in the Aerospace, Power, Medical Imaging, and Oil & Gas domains. To address the growing scale and complexity of these critical software intensive systems, GE Global Research, in collaboration with GE Aviation Systems, developed a tool named ASSERT™ (Analysis of Semantic Specifications and Efficient generation of Requirements based Tests) [1, 2]. ASSERT™ helps users capture requirements that are both human and machine-readable using structured natural language. Having a requirements language that is as close as possible to English, instead of a language that is directly based on first-order logic, temporal logic, or some other symbol-heavy formalism, is the entry point for convincing users to adopt the tool into their design process. ASSERT™ checks the requirements for conflicts and completeness by translating the structured natural language into a formalism that an automated theorem prover can reason over. It also translates the requirements into satisfiability modulo theories (SMT) formulas in such a way that an SMT solver can automatically generate requirements-based test cases. ASSERT™ saves time and cost by identifying errors early in the development process and automating requirements-based test generation.

The focus of this industrial experience paper is to highlight common problems that designers and engineers experience when capturing requirements, and to demonstrate how the application of formal methods using ASSERT™ helps resolve these issues. The application space selected for demonstration is a sensor used in the aerospace domain. The sensor use case was selected as it is a clear and simple example that readers will understand and is relevant in many diverse applications.

The paper is organized as follows. Section II gives a brief history and rationale for development of the ASSERT™ tool. Section III describes typical problems encountered while capturing requirements. Section IV introduces a semantic model for an aircraft engine sensor. In section V, we illustrate authoring of the requirements in ASSERT™ and how some common requirement capture pitfalls are addressed. In Section VI we present the automated formal analyses of the requirements with user-friendly reporting. In Section VII, we discuss test case generation and SCADE simulation-based test execution for the sensor example. In Section VIII we discuss some of the lessons learned. In Section IX we present our conclusions. The complete aircraft engine sensor model and requirements are available in the Appendix.

II. Background

GE Aviation Systems (GEAS) produces avionics that manage and control safety critical functions on aircraft such as integrated modular avionics and flight management systems. The level of effort required for development and certification of these systems is very high and growing based on increased software and system complexity. In 2010, GE Aviation Systems and GE Global Research initiated a research program to develop technology and a new process to increase efficiency, improve safety, and reduce cycle time to develop and qualify safety critical aerospace software and systems. The program focused on formal modeling and analysis of requirements during the design process to identify and address errors and vulnerabilities early. The program included the use of structured natural language requirements capture, design modeling, auto test case generation, and formal methods to efficiently satisfy

DO-178C [3] software development guidance and verification objectives. At this point the program has developed the ASSERT™ suite of tools and has provided them to product teams through the GEAS tools dissemination organization.

ASSERT™ is currently being used on several projects within GE and has been shown to be capable of handling industrial size problems. To date these projects have mostly involved functional requirements. In one avionics project, use of the tool for iterative correction and refinement culminated in a set of 1002 requirements with no errors from which 19,276 test cases and 38,272 test procedures were automatically generated. There were several noted benefits on another project for an avionics compute platform with over 500 requirements. First, ASSERT™ helped produce requirements that were higher in maturity than plain text requirements due to the error checking feature in requirements capture and the requirements analysis. Second, the system design effort was reduced because the ontology developed as part of the requirements capture already provided a functional architecture. Lastly, using ASSERT™ to automatically generate test cases from requirements was very beneficial as part of the test-driven development process because it ensured that requirements are verifiable as soon as they were written. In earlier papers [1, 2] we introduced ASSERT™ and details of the analysis output. In this paper we expand on the theoretical underpinnings of the formal analysis and on test cases and procedures.

III. Requirement Pitfalls

Writing requirements is not an easy task; see for example [4]. The requirements writer faces multiple problems which we will broadly classify as 1) determining what the system should do and 2) capturing that specification as unambiguously as possible. Ideally a tool will help with both challenges and in practice the process is always highly iterative. As the requirements author strives to capture what is required she gains deeper insights into what is required, leading to further efforts to refine the specification. We will now examine some of the difficulties encountered when writing requirements.

A. Ambiguities

Most people find their native tongue to be the easiest language in which to describe something. Ease of expression is important because the more mental resource allocated to the effort of expression, the less mental resource available to think about what is being expressed. However, natural language is famously vague and ambiguous, the opposite of what is desired in requirements. This creates a tension between ease of expression on the one hand and precision of meaning on the other. We have sought a balance between these competing needs by creating a controlled English requirements language. However, requirements must be specified in domain terms, and a domain-specific language for requirements expression is derived from a domain ontology that defines the relevant concepts. Both the ontology language and the requirements language map unambiguously into a formal representation which is a subset of first order logic and set theory. This makes requirements amenable to formal methods for analysis and for automatic test case generation.

A design goal of the ASSERT™ requirements authoring environment is to provide as much useful feedback as possible to the user as quickly as possible. This can be in the form of an error marker indicating that something is definitely wrong and must be resolved before any further analysis is possible. Sometimes a statement in a requirement will be marked with a warning indicating that while it is not strictly incorrect, statements of this type are often not what the author intended and care should be taken. Some markers are informational only, informing the author that a particular assumption was made about what was expressed. Errors, warnings, and informational markers are used to make the author aware of errors and ambiguities in what has been written and assumptions that have been made to avoid ambiguity.

B. Separating Levels of Detail

Requirements are often broken down into levels ranging from system requirements to high-level requirements to low-level requirements. Regardless of the level, the distinction is often made between requirements that specify what a system is expected to do and the design of how the system is to accomplish the what. Requirements are about the what. Design models are about the how [5].

A requirement capture environment must somehow allow the author to not say everything in high-level requirements but rather indicate that additional detail will be provided in lower-level requirements. In ASSERT™ this is accomplished through decomposition. A decomposition is an incomplete specification. For example, a decomposition might include descriptive text that documents informally what should happen and is to be specified in lower-level requirements or design.

C. Inconsistent References to Types or Units

An ontology captures a model not only of the types of things (classes) relevant to a domain, but also captures the kinds of relationships which can exist (properties, also known as relationships and attributes). It is often the case that a property definition will include the kinds of things which can have that property (the domain of the property) and the kinds of values that the property can have (the range of the property).

It is a very common pitfall that when a requirements writer begins to specify what the system under design shall do, these statements are not consistent with the property definitions and restrictions in the ontology. Checking to make sure that all statements in the requirements are consistent with the ontology is called type checking. For example, type checking would generate an error marker if a statement indicated that a Rock was to be inserted into a list of Food items. Type checking would also generate an error if a thing was to be inserted into something which is not a list. Type checking ensures that the requirements make sense at a basic consistency level.

Writing requirements that satisfy type checking can be very demanding. In normal conversation, we might say that “He is 45.” If he is an adult, this might mean “His age is 45 years” or if he is a child that “His weight is 45 pounds.” We say the shorter version because it is normally, in context, unambiguous and more parsimonious; and parsimony is important to make requirements more easily understood. In ASSERT™ we allow a limited form of the same parsimony using implied properties. If age is a property with domain person and range decimal, and if age is an implied property of person, then the statement “The person is 45” will be automatically interpreted as “The age of the person is 45” provided there is only one implied property of person with range matching the numeric type of 45. Implied properties allow the expression of requirements to be closer to how subject matter experts think about a domain.

Especially in lower-level requirements, units matter. Failures caused by mismatched units have downed airplanes, sent space flights off course, and crashed vehicles [6]. Inclusion of units on numbers and expressions is supported in ASSERT™.

D. *Assumptions Not Captured; Conflicting or Incomplete Requirements*

For large projects, the system is typically broken down into subsystems and the design process involves multiple developers. One subsystem may depend on another and sometimes developers neglect to properly capture these dependencies. ASSERT™ allows for the capturing of these types of assumptions. For example, an assumption might capture the range of values expected for a property based on another subsystem or even on the environment.

It is also not possible for everyone on the team to understand all the details of the entire project. This can result in a developer having an incorrect or incomplete mental model of the entire system. Resulting conflicts and incompleteness issues are typically caught during testing and code coverage analysis, which happens later in the design process and hence any issues found can lead to extensive re-work.

Our approach in ASSERT™ is to catch these issues as early in the process as possible. We do this by employing explainable and automated formal analysis of the requirements. The requirements engineer does not need to be well-versed in formal analysis because the formal analysis in ASSERT™ is automated and the output is explainable. ASSERT™ localizes an issue and provides counterexamples in terms of the concepts and properties used in the requirements. These counterexamples focus the attention of the requirements engineer on the exact issue to be resolved.

IV. Example of an Aircraft Engine Sensor

We have modeled parts of an aircraft engine and related requirements. In this paper, we will use a simplified pressure sensor as a running example. There are numerous publications that describe how to handle redundant sensor selection, such as PS3 and T5, in aircraft engines controls [7]. Here we will describe a dual-channel pressure sensor *Px* as specified in Fig. 1, where the selected value is based on the inputs received.

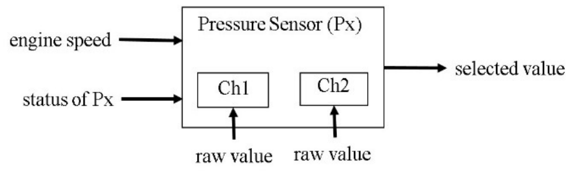


Figure 1. Dual channel pressure sensor.

Engine speed is used as a proxy for determining the status of the aircraft and is used to set range limits on pressure values. *Px* has two channels, *Ch1* and *Ch2*, and each channel receives an input raw signal. The status of *Px* is also an input to *Px* – this is a signal indicating the status of *Px* as determined by other mechanisms. ASSERT™ captures the semantic model in the controlled English Semantic Application Design Language (SADL) [8,9]. High-level details of *Px* are as follows.

```
SYSTEM is a class.
Engine is a class.
engineSpeedPercentage describes Engine
    with a single value of type decimal.

Sensor is a class.
statusOk describes Sensor with a single value of type boolean.
DCPS is a type of Sensor.    // DualChannelPressureSensor
channels describes DCPS with values of type TwoChannels.

TwoChannels is a type of Channel, must be one of {Ch1, Ch2}.
Px is a type of DCPS.        // Pressure Sensor
Channel is a class.
```

This describes a sparse skeleton that can be expanded upon by adding additional properties and restrictions. Note that the class-subclass structure provides inheritance of properties and so, for example, the *statusOk* property is defined for *Px,* a subclass of *Sensor*. Notice the readability of the model as specified in ASSERT™.

The flow of processing for the pressure sensor *Px* is as follows. The raw signals for each channel are provided in mV units and are converted to psi units and filtered. For fault detection and isolation, we will follow the traditional approach of determining hard faults for each channel based on specified limits and persisted hard faults become hard fail for the channels [10]. At the pressure sensor level, soft fault is identified based on the difference in values of the converted values of the two channels. A persisted soft fault becomes a soft failure. The pressure sensor is declared persistent bad if both of its channels have hard fail and the pressure sensor has a soft fail.

We now expand on the skeleton ontology presented above by adding the additional properties we will need in capturing the requirements for *Px* including defining unitted quantities.

```
selectedValue describes DCPS with a single value of type decimal.
persistentBad describes DCPS with a single value of type boolean.
softFault describes DCPS with a single value of type boolean.
softFail describes DCPS with a single value of type boolean.
lowRangeLimit describes DCPS with a single value of type decimal.
highRangeLimit describes DCPS with a single value of type decimal.
rawValue describes Channel with a single value of type decimal.
convertedValue describes Channel
   with a single value of type decimal.
filteredValue describes Channel with a single value of type decimal.
hardFault describes Channel with a single value of type boolean.
hardFail describes Channel with a single value of type boolean.
```

The selected value of the pressure sensor is then determined as specified by the flow chart in Fig 2.

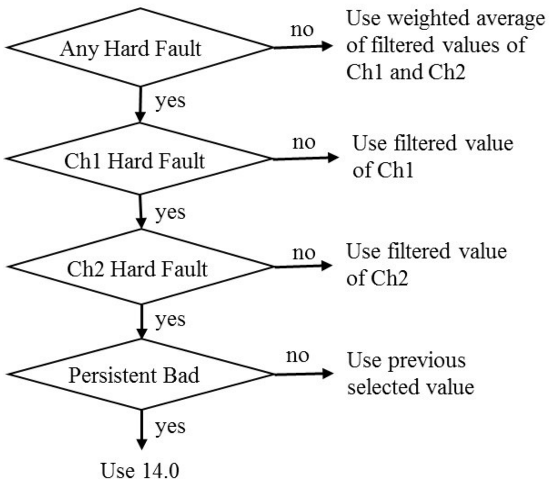

Any Hard Fault
no
Use weighted average of filtered values of Ch1 and Ch2
yes
Ch1 Hard Fault
no
Use filtered value of Ch1
yes
Ch2 Hard Fault
no
Use filtered value of Ch2
yes
Persistent Bad
no
Use previous selected value
yes
Use 14.0

Figure 2. Determination of selected value.

This section highlighted part of the ontology used; the full ontology and requirements for *Px* are given in the Appendix.

V. Formal Requirements Capture

To perform formal analysis on requirements, we need to first formally capture the requirements. The requirements are captured in the SADL Requirements Language (SRL). We present some basic requirement definitions relevant to ASSERT™ and used for the sensor example. A more in-depth technical discussion on the development of a requirements language from an ontology language can be found in [1, 2].

There are other tools that capture requirements and analyze them. Goals are like ours – ease an engineer's transition from natural language to formal requirements; minimize the semantic distance between the subject matter expert's mental model and the target language. RSML (Requirements State Machine Language) [11] is one such early work, including its follow-on research SpecTRM (Specification Tools and Requirements Methodology) [12], designed primarily for process-control systems and is based on state machines. SpeAR (Specification and Analysis of Requirements) [13, 14] is another tool for capturing and analyzing requirements. It is open source and has a specification language front-end that gets translated to a Lustre model and analyzed for logical conflicts, among other checks. Our tool differs in our preference for ontology-based technology. The core of ASSERT™ is the ability to capture the domain as a semantic model. The semantic model based on SADL, which we developed as a domain-specific language for representing ontologies in the Web Ontology Language (OWL). SADL also provides if-then rules. SADL is both a controlled-English language and a rich integrated development environment (IDE) or modeling environment for creating, testing, and maintaining semantic models. We feel that the selection of semantic modeling is superior to other technologies for a variety of reasons. High among them is the modularity and extensibility that are achieved through a formal semantics based on set theory. Set theory encompasses inheritance and inheritance allows information about a model concept to be captured in a single model location but used in as many locations as appropriate.

A. Anatomy of a Requirement

A requirement is a statement made up of three main components: a name, a conclusion, and a condition. The name is a unique identifier given by the user or assigned by a requirements management tool such as DOORS®. The conclusion is the *shall* part of the statement that specifies something that is required to be true. The condition is the *when* part which specifies the applicability of the requirement. Here is an example.

```
Requirement LowRangeLimit1:                              // name
SYSTEM shall set lowRangeLimit of Px to 1.0              // conclusion
when engineSpeedPercentage <= 30.0 .                     // condition
```

lowRangeLimit and *engineSpeedPercentage* are properties and are defined in the ontology model. Each concept has a single meaning shared across all users. In each requirement, we differentiate between properties whose values are being set in the conclusion, which we refer to as "controlled properties" or "controlled variables," and properties whose values are used in the condition, which we refer to as "monitored properties" or "monitored variables." Note that it is certainly possible for a property to be a controlled variable in one requirement and be a monitored variable in another.

B. Compact Requirements based on Context and Grounding

The controlled English requirements language in ASSERT™ allows for a compact representation. The user can refer to a property without explicitly spelling out its full class path by specifying a *Context*. The *Context* sets the focus for a set of requirements and limits the scope of interpretation so that full class paths are not needed to avoid ambiguity. Suppose the property name *statusOk* is used in multiple classes, such as *P0* and *Px*. If we set the *Context* in a requirements file as follows

```
Context: Px with channels TwoChannels.
```

then in that file we can use *statusOk* instead of *statusOk of Px*.

The user can also write requirements at the class level and grounding expands the requirement for each instance of a class. Here is an example that illustrates grounding.

```
Requirement HardFault:
SYSTEM shall set hardFault of channels of Px to true
only when convertedValue of channels of Px < lowRangeLimit of Px
      or convertedValue of channels of Px > highRangeLimit of Px.
```

Without grounding, we would need two requirements, one for *Ch1* and one for *Ch2*. ASSERT™ internally expands out *HardFault* requirement into two requirements.

C. Continuous Type Checking

As requirements (and ontology) are authored in ASSERT™, type checking is performed continually and errors are flagged as illustrated by the following requirement.

```
Requirement SoftFault1:
SYSTEM shall set softFault of Px to true
when abs(convertedValue of Ch1 - convertedValue of Ch2) > 15.0
        and statusOk of Px is true.
```

The property *softFault* is defined in the ontology model as a boolean. This prevents a user from writing a requirement like the following.

```
Requirement SoftFault1:
SYSTEM shall set softFault of Px
to abs(convertedValue of Ch1 - convertedValue of Ch2)
when statusOk of Px is true.
```

ASSERT™ will immediately flag this an as error, notifying the user that *softFault* takes a boolean value and cannot be compared with an expression of type decimal.

D. Unitted Quantity

In addition to type checking, ASSERT™ also handles unitted quantity. This makes for even stronger typed requirements in that not only do primitive types must agree, but their units also must be consistent. For example,

```
rawValue describes Channel with a single value
   of type UnittedQuantity.
convertedValue describes Channel with a single value
   of type decimal.
Signal_mV is a type of UnittedQuantity.
unit of Signal_mV always has value "mV".
rawValue of Channel only has values of type Signal_mV .
```

In our example, we use unitted quantity to specify when a property is in millivolt (mV). Unit conversion errors can lead to egregious miscalculations. Unitted quantity prevents the user from writing requirements that sets/compares *rawValue* to/with *convertedValue*.

E. Specifying What and Not How by Using Decomposition

Decomposition in ASSERT™ allows the requirements writer to say what a system is expected to do without being forced to specify the implementation details. This allows the user to stay at the right level of abstraction. Suppose in the ontology we define

```
SecondOrderTustinFilter is a class.
inp describes SecondOrderTustinFilter with a single value
   of type decimal.
_value describes SecondOrderTustinFilter with a single value
   of type decimal.
```

We use *_value*, as *value* is a reserved word in ASSERT™. Now without giving further details such as the gains or the update rate for *SecondOrderTustinFilter*, we can use it as a decomposition as follows.

```
Requirement FilteredValue:
SYSTEM shall set
filteredValue of channels of Px to _value of filteredSignal
where filteredSignal is (a SecondOrderTustinFilter
          with description "2nd order Tustin with K=1, D=0.5, N=0.5"
          with inp (convertedValue of channels of Px) ).
```

We only need to specify the values for required inputs. A special property, *description*, allows us to capture text that can be used when a designer implements the decomposition. Note also the use of *where* clause that allows us to introduce and use a local variable (*filteredSignal*).

F. Equations

In contrast to decomposition where details are abstracted away, equations allow the user to capture an exact detail for implementation. This feature is important for recording conversion formulas from a supplier or for documenting how we shall compute an output used by a customer downstream. In our example, we use the following equation for converting the raw pressure sensor signal to engineering units.

```
Equation Conversion(decimal mult, decimal raw-input,
                              decimal adder) returns decimal:
mult * raw-input + adder.
```

Equations are analyzed in a similar way to decomposition. Analysis can be proven or disproven, depending on the set of requirements analyzed.

VI. Formal Requirements analysis

Requirements are formally analyzed using RAE, the patented [15] Requirements Analysis Engine of ASSERT™. RAE can be used to analyze requirements as soon as they are written, without the need for lower-level requirements, annotations, properties or code. RAE can also be used to analyze an incomplete set of requirements, which allows requirements engineers to get meaningful feedback immediately. If errors are found, RAE will localize the error by identifying which requirements are responsible, it will generate a test case showcasing the error, where appropriate, and it will generate a report explaining the error. RAE errors never include false alarms, i.e., RAE generates error reports only when it has a proof that an error exists.

To formally analyze SRL requirements we need to translate them to a formal language with mechanized logic and theorem proving and disproving support. RAE uses the ACL2 logic as its base language and the ACL2s system as its background solver. ACL2s, the ACL2 Sedan [16], is itself an extension of the ACL2 theorem prover, featuring automated termination analysis [17, 18], counterexample generation capability [19, 20] and a convenient typed language support [21].

In this section, we briefly outline the core technical details of how RAE works. First, we informally describe how major aspects of the SADL ontology and SRL requirements map to a syntactic extension of the ACL2s typed language [21]. Then, we describe how conflict and completeness analysis are mechanized in ACL2s. We assume some familiarity with ACL2, an industrial-strength interactive theorem prover that has been successfully applied to verify complex hardware designs. We refer the reader to [22] for a short introduction and to [23] for a more complete description.

A. Translation to ACL2

The requirements engineer captures the concepts and the relational vocabulary of the system under design in a SADL ontology. Concepts are captured as SADL classes and relations as SADL properties. The requirements specifying the behavior of the system under design are captured in SRL.

1) Translating Ontology.

Primitive SADL datatypes such as decimal and boolean are translated directly to corresponding basic types in ACL2s. Each SADL class *C* gets translated to an ACL2s data definition form, *(defdata C <def>)*. The *defdata* form concisely supports common data definition patterns, e.g. list types, enum types, record types, range types, etc. The *defdata* form defining *C*, among other things, automatically generates a predicate function (*Cp*) and an enumerator function (*nth-C*) that recognize and generate values of type *C*.

Range restrictions on primitive datatypes are translated to *defdata* range types. Many SADL classes are explicit enumerations and are encoded using enum types. Undefined SADL classes are translated to concrete enum types by taking into account their subclasses and occurrences of property expressions of that type. List classes with length constraints are translated to list types with appropriate length restrictions.

2) Translating Requirements.

Below, we assume that SRL requirements have been grounded as described in a previous section. Each SRL requirement form is translated to a *defrequirement* form with the following structure:

```
(defrequirement name
  ((mvar1 Cp1) ..)
  ((cvar Cp))
  (implies when_formula
       (equal cvar rhs_expression)))
```

The first element of the form is the name of the requirement. The second is a list of monitored variables along with their types. The third is a list containing a single controlled variable and its type. The last element of the *defrequirement* form, called its body, is an ACL2 formula.

The translation is as follows. The property expression, *lhs*, in the *shall set lhs to rhs* SRL construct is translated to a fresh controlled variable, e.g., *cvar* in the above form. Suppose the range of this property is a SADL class (or primitive SADL datatype) that is translated to *defdata* type *C*, then we associate *cvar* with its type predicate *Cp*. All other property expressions are encoded as monitored variables along with their respective types as above. The *shall set lhs to rhs* construct is translated to an equality whose first argument is *cvar* and whose second argument is a translation of the SRL expression *rhs*. If the when condition is present, the requirement body is encoded as an implication whose antecedent is a translation of the when condition expression and whose conclusion is the equality expression described above.

We note that the *defrequirement* form itself macro-expands to an ACL2s *defunc* form that defines functions along with its input-output contract (i.e. type signature). Each top-level SRL construct (including assertion, assumption and equation) is translated to a form that has a similar structure as *defrequirement*. Occurrences of decomposition expressions such as

```
(a C with p1 (pexp1) with p2 (pexp2) ...)
```

give rise to *defdecomposition* forms that define an uninterpreted function whose signature is determined by the translated types of class *C* (output) and properties *p1,p2,..* (inputs). The decomposition expression is translated to the corresponding function call whose arguments are the translations of the property expressions, *pexp1*, *pexp2* appearing in the decomposition.

ASSERT™ also handles qualitative and quantitative timing constraints. Timing conditions are translated away into regular ACL2 boolean expressions using fresh copies (past snapshots) of monitored variables occurring in the conditions. The previous operator that can be used as a prefix to any property expression in SRL gives rise to a fresh monitored variable. The remaining SRL expressions map to ACL2 expressions, using a syntax-directed translation.

B. Formal Requirements Analysis using ACL2s

All requirements, assumptions, assertions, equations pertinent to a single controlled variable are collected together and translated as a unit. RAE does a sequence of formal analyses on each such translated unit. The first analysis RAE performs is type-safety analysis. All operators, equations, decompositions are typed and the pervasive use of *defdata* and *defunc* forms program the ACL2s typechecker (Tau-system) to automatically discharge type-like proof obligations [21]. Among the other analyses, two major ones RAE performs are conflict and completeness, which we describe below.

RAE performs conflict and completeness analysis by generating a corresponding ACL2 formula and querying its validity using the ACL2 theorem prover and its falsifiability using ACL2s counterexample generation facility, Cgen [19, 20]. Cgen uses a synergistic combination of property-based testing and theorem proving to find counterexamples to ACL2 conjectures. The enumerators that perform property-based test data generation are automatically synthesized by the *defdata* form.

When the proof is found, RAE reports that the analysis passed. If a counterexample is found, RAE reports an analysis error. Note that although we have carefully programmed ACL2s to achieve automation and minimize the number of undetermined results, i.e., ACL2s is neither able to prove the query, nor is ACL2s able to falsify it, our method is not a decision procedure and we sometimes get undetermined results that are reported as potential errors. The formal details, including checks for conflicts, completeness, contingency, independence and surjectivity, are described in detail elsewhere [15].

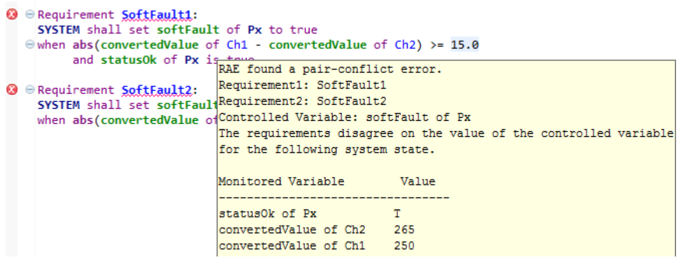
In a typical project, requirements analysis is done iteratively as requirements are added and modified. For the project with 1002 requirements mentioned earlier, the processing time for a complete requirements analysis was under half an hour. The requirements analysis is also incremental which can significantly reduce requirement analysis processing time as requirements are added and modified.

C. Illustration of Requirements Analysis

We will now illustrate formal analysis for the sensor example. For *Px*, we require that *softFault* be set to true only when the difference between the converted values of the channels is greater than 15, provided the *statusOk of Px* is true. Suppose the following requirements were drafted.

```
Requirement SoftFault1:
SYSTEM shall set softFault of Px to true
when abs(convertedValue of Ch1 – convertedValue of Ch2) >= 15.0
        and statusOk of Px is true.
Requirement SoftFault2:
SYSTEM shall set softFault of Px to false
when abs(convertedValue of Ch1 – convertedValue of Ch2) <= 15.0.
```

This is an example of a common error – did we want the difference to be strictly greater than 15 or greater than or equal to 15? RAE reports the error message as shown in Fig. 3 that includes a counterexample (test case). Note that the counterexample captures the system state where the gap between the converted values is exactly 15, in which case both requirements apply and set *softFault of Px* to different values (true is represented as *T*). There is a conflict among the requirements! This feedback is presented contextually with the offending requirements that makes it easy to understand and correct.

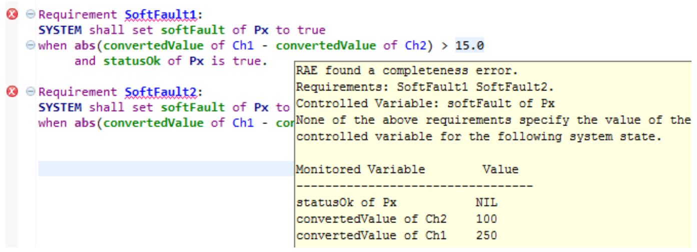


```
Requirement SoftFault1:
SYSTEM shall set softFault of Px to true
when abs(convertedValue of Ch1 - convertedValue of Ch2) >= 15.0
     and statusOk of Px is

Requirement SoftFault2:
SYSTEM shall set softFaul
when abs(convertedValue o
```

```
RAE found a pair-conflict error.
Requirement1: SoftFault1
Requirement2: SoftFault2
Controlled Variable: softFault of Px
The requirements disagree on the value of the controlled variable
for the following system state.

Monitored Variable        Value
---------------------------------
statusOk of Px            T
convertedValue of Ch2     265
convertedValue of Ch1     250
```

Figure 3. Illustration of a pair-conflict error with counterexample.

The requirements engineer corrects her mistake and replaces the >= sign in the first requirement with the > sign. This resolves the conflict, but now RAE reports another error as shown in Fig. 4.



```
Requirement SoftFault1:
SYSTEM shall set softFault of Px to true
when abs(convertedValue of Ch1 - convertedValue of Ch2) > 15.0
     and statusOk of Px is true.

Requirement SoftFault2:
SYSTEM shall set softFault of Px to
when abs(convertedValue of Ch1 - cor
```

```
RAE found a completeness error.
Requirements: SoftFault1 SoftFault2.
Controlled Variable: softFault of Px
None of the above requirements specify the value of the
controlled variable for the following system state.

Monitored Variable        Value
---------------------------------
statusOk of Px            NIL
convertedValue of Ch2     100
convertedValue of Ch1     250
```

Figure 4. Illustration of completeness error with counterexample.

The difference between the channel values is greater than 15, but since *statusOk* is false (represented as *NIL*), no requirement applies, and RAE displays a system state where the behavior of the system is unspecified. The requirements are not complete. The requirements engineer can correct this by adding a third requirement that specifies what is to be done in the case *statusOk* is false. With the third requirement added, RAE reports no errors, asserting that the three requirements for controlled variable *softFault of Px* are complete and conflict-free.

VII. AUTOMATED TEST CASE GENERATION

After requirements are analyzed with RAE, they are then processed by ATG, the Automated Test Generation tool of ASSERT™. ATG automatically analyzes the requirements against predefined test coverage criteria (based on DO-178C guidelines) and produces requirements-based test cases and procedures. Here we provide a brief description; some additional details can be found in [1, 24-26].

Prior work in this area includes [27-29] that discuss test case generation from design models, such as SCADE, to achieve structural coverage at the design/code level. Our automated test generation tool focuses on generating test cases from requirements written in structured natural language to achieve test coverage at the requirements level. The generated test cases are then executed on the design model/code to reveal inconsistencies between requirements and design/code. Unlike the prior work, our tool's ability to generate test cases before a single line of code is written promotes independence between the person generating the requirements-based test cases and the developer of the code to be verified.

A. Test Case Generation

Formal methods are applied in ATG to perform key tasks, such as reachability analysis, test optimization, test sequencing, test data generation, and initial condition setting. ATG first automatically derives test objectives from requirements and test coverage criteria, then for each of the above tasks ATG synthesizes a set of satisfiability modulo theories (SMT) formulas from the test objectives and applies SMT solvers to check the satisfiability of the formula. The solver results, including the counterexamples, are used for removing invalid test cases, combining redundant test cases, generating satisfiable input/output data, etc. The generated test cases are formatted to support human review. They are written in a text file with tables to show how each test coverage criteria are satisfied by the generated test cases. The test case files also provide justifications for when a certain coverage criterion is not achieved. Test procedures are generated separately. The test procedures are machine readable. They describe step-by-step instructions on how to set the initial conditions and inputs to verify the actual outputs against the expected outputs.

B. SCADE Simulation

The test procedures can then be automatically mapped to the target test platform and be executable against the system or software under test. The Automated Test Procedure Translator tool of ASSERT™ takes the test procedures generated by ATG, a template describing the format of the tests in the target test environment, and a mapping of names from how they appear in ASSERT™ to how they appear in the test environment, and generates test scripts. These test scripts can then be directly executed in the target test environment using third-party tools for functional verification and test coverage measurement. Note that the software under test is an output of a separate development activity and not directly part of ASSERT™.

We now show how the test procedures are simulated against a SCADE design model. Fig. 5 shows an implementation that meets the requirements for the selected value of *Px*. Fig. 6 is the compilation of the test cases run in the SCADE simulation environment. Fig. 7 shows graphically the test inputs along with the output that can be validated against the expected values from ATG.

VIII. Lessons Learned

There are several lessons learned when designing a tool for industry-wide acceptance. For business leaders to use a new requirement capture and analysis tool, they must be convinced that the engineers can be trained to use the tool. It is unlikely that we could have sold that argument if the language used was directly based on first-order logic, temporal logic, or some other symbol-heavy formalism. By showing management and engineers that ASSERT™ requirements look very similar to the requirements they were writing in English, we were able move beyond this first hurdle.

To be accepted, a new requirement capture and analysis tool must offer means of mitigating the risks of adoption. Since the ASSERT™ language is English-like, it was possible to argue that even if requirements analysis and test case generation were not fully successful, there was still much benefit in capturing requirements in ASSERT™, e.g., having a common ontology and automated type checking, all in an integrated system, were on their own worth it.

Having engineers from the business units embedded into the team developing the tools was important because it kept us focused on solving the real problems the business was having. We also developed a deep bench of power-users in the business who have the domain expertise to explain how to effectively use the tool to others in the business, and finally we gained access to real problems. All of this allowed us to make the value proposition to management, which was needed to fund a multi-year, multi-team project like this.

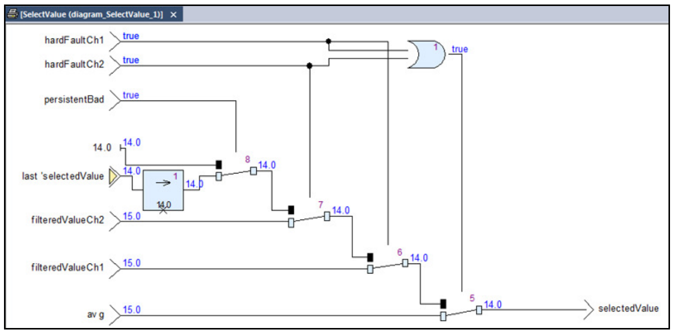


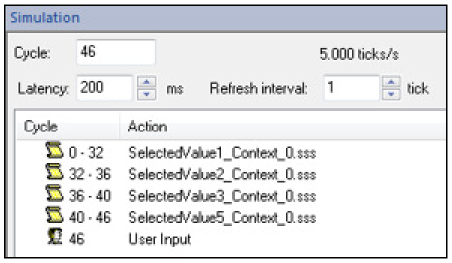
Figure 5. SCADE implementation.

IX. Conclusions

In this paper we have listed some of the common requirement pitfalls that inadvertently introduce errors into a program. Many studies have shown that the cost of fixing errors in the designs of safety-critical systems depends on the time elapsed between error introduction and error discovery. As the gap between error introduction and error discovery increases, so does the cost of resolving such errors. The resolution may require architectural changes and may have a cascading effect that requires redesigning various aspects of the system, which leads to delays and cost overruns. In this paper, we have shown how ASSERT™ helps a developer capture complete and conflict free requirements, and provides explainable and automated formal analysis to identify and help resolve errors during the requirements authoring process. Of notable importance is the fact that the domain experts using the ASSERT™ tool do not need to be formal modeling or analysis experts to benefit from the formal methods used to assist them in the capture, analysis and automated test generation of their requirements.

Acknowledgment

The authors gratefully acknowledge Scott Stacey, Tony Lanzillotti, Jeff Gilton, Shreecharan Kanchanavally, Craig McMillan, Gary Quackenbush, Christin Rauche, and Michael Idelchik for their unwavering support throughout the course of this research.


Simulation

Cycle: 46 5.000 ticks/s

Latency: 200 ms Refresh interval: 1 tick

Cycle	Action
0 - 32	SelectedValue1_Context_0.sss
32 - 36	SelectedValue2_Context_0.sss
36 - 40	SelectedValue3_Context_0.sss
40 - 46	SelectedValue5_Context_0.sss
46	User Input

Figure 6. Compilation of the test cases run in the SCADE simulation environment.

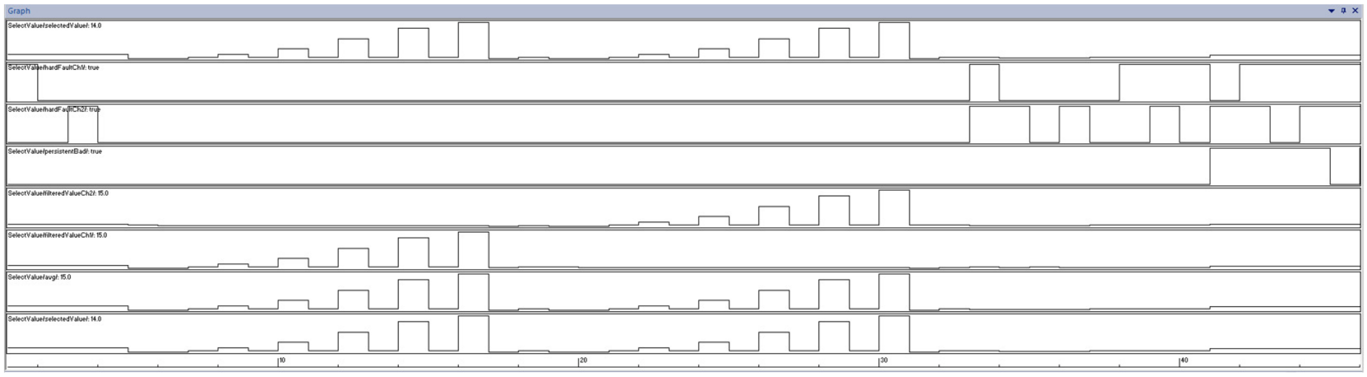


Figure 7. Test inputs along with the output that can be validated against the expected values from ATG.

References

[1] Siu, K. *et al.*: Flight critical software and systems development using ASSERT™. In: IEEE/AIAA 36th Digital Avionics Systems Conference (DASC), pp. 1-10, St. Petersburg, FL, USA (2017).

[2] Crapo, A., Moitra, A., McMillan, C., Russell, D.: Requirements Capture and Analysis in ASSERT™. In: IEEE 25th International Requirements Engineering Conference (RE), pp. 283-291, Lisbon, Portugal (2017).

[3] DO-178C Software Considerations in Airborne Systems and Equipment Certification. RTCA, 12/13/2011.

[4] Brat, G., Bushnell, D., Davies, M., Giannakopoulou, D., Howar, F., Kahsai, T.: Verifying the safety of a flight-critical system. In: International Symposium on Formal Methods, pp. 308-324. Springer, Cham. (2015).

[5] Whalen, M., Gacek, A., Cofer, D., Murugesan, A., Heimdahl, M., Rayadurgam, S.: Your "What" is My "How": Iteration and Hierarchy in System Design, IEEE Software, 30 (2), (2013).

[6] Some Famous Unit Conversion Errors. [Online] https://spacemath.gsfc.nasa.gov/weekly/6Page53.pdf

[7] Litt, J.S., Simon, D.L., Garg, S., Guo, T.H., Mercer, C., Millar, R., Behbahani, A., Bajwa, A., Jensen, D.T.: A survey of intelligent control and health management technologies for aircraft propulsion systems. JACIC, 1 (12), 543-563 (2004).

[8] Semantic Application Design Language (SADL). [Online] http://sadl.sourceforge.net/index.html.

[9] Crapo, A. Moitra, A.: Toward a unified English-like representation of semantic models, data, and graph patterns for subject matter experts. International Journal of Semantic Computing, 7 (3), 215-236 (2013).

[10] Laprie, J. C., Arlat, J., Beounes, C., Kanoun, K.: Definition and analysis of hardware-and software-fault-tolerant architectures. Computer, 23 (7), 39-51 (1990).

[11] Leveson, N., Heimdahl, M., Hildreth, H., Reese, J.: Requirements specification for process-control systems. In: IEEE Transaction on Software Engineering (1994).

[12] Leveson, N., Heimdahl, M., Reese, J.: Designing specification languages for process control systems: lessons learned and steps to the future. In Software Engineering – ESEC/FSE '99. Lecture Notes in Computer Science, vol 1687. O. Nierstrasz, M. Lemoine (eds). Berlin, Heidelberg: Springer (1999).

[13] GitHub SpeAR. [Online] https://github.com/lgwagner/SpeAR.

[14] Wagner, L., Fifarek, A., DaCosta, D., Gross, K.: SpeAR: Specification and Analysis of Requirements. In: S5 Symposium (2014).

[15] Manolios, P.: Scalable methods for analyzing formalized requirements and localizing errors. U.S. Patent 9,639,450, May 2, 2017.

[16] Chamarthi H.R., P.C. Dillinger, P.C., Manolios P., Vroon D.: The ACL2 Sedan theorem proving system. TACAS, 2011, Springer.

[17] Manolios, P. Vroon, D.: Termination analysis with calling context graphs. Computer Aided Verification (CAV), Lecture Notes in Computer Science 4144, Springer, pp. 401–414 (2006).

[18] Manolios P., Vroon, D.: Interactive termination proofs using termination cores. Interactive Theorem Proving, July 2010, Springer LNCS 6172.

[19] Chamarthi, H.R., Dillinger, P.C., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving. In: ACL2 2011, EPTCS 70, pp. 4–19.

[20] Chamarthi, H.R., Manolios, P.: Automated specification analysis using an interactive theorem prover. In: FMCAD 2011, pp. 46–53.

[21] Chamarthi H.R., Dillinger P.C., Manolios, P.: Data Definitions in the ACL2 Sedan. ACL2 pp. 27-48 (2014).

[22] ACL2 Tutorial. [Online] http://www.cs.utexas.edu/users/moore/acl2/v7-4/combined-manual/

[23] Kaufmann, M., Manolios, P., Strother Moore J.: Computer-aided reasoning: an approach. Kluwer Academic Publishers (2000).

[24] Li, M.: Integrated automated test case generation for safety-critical software. U.S. Patent Application 20160170864A1, filed December 2014.

[25] Li, M., Durling, M., Siu, K., Oliveira, I., Yu, H., De Conto, A.: System and method for safety-critical software automated requirements-based test case generation. U.S. Patent 9,940,222, April 10, 2018.

[26] De Conto, A., Li, M., Manolios, P., Oliveira I.: System and method for equivalence class analysis-based automated requirements-based test case generation. U.S. Patent Application 20170228309A1, filed February 2016.

[27] Durrieu, G., Laurent, O., Seguin, C., Wiels, V.: Formal proof and test case generation for critical embedded systems using SCADE. In: Building the Information Society, vol. 156, pp. 499-504, R. Jacquart, Eds. Boston, MA: Springer (2004).

[28] Wiels, V., Delmas, R., Doose D., Garoche, P.L., Cazin, J., Durrieu, G.: Formal verification of critical aerospace software. In: Aerospace Lab, pp. 1-8 (2012).

[29] Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V.: Model checking flight control systems: the Airbus experience. In: International Conference on Software Engineering (2009).

X. Appendix: Domain Model and Requirements

The semantic domain model is as follows.

```
uri "http://sadl.org/PressureSensor" alias PS.

SYSTEM is a class.

Engine is a class.
engineSpeedPercentage describes Engine
    with a single value of type decimal.

Sensor is a class.
statusOk describes Sensor with a single value of type boolean.

Signal_mV is a type of UnittedQuantity.
unit of Signal_mV always has value "mV".

Signal_psia is a type of UnittedQuantity.
unit of Signal_psia always has value "psia".

DCPS is a type of Sensor. //DualChannelPressureSensor
selectedValue describes DCPS
    with a single value of type decimal.
persistentBad describes DCPS
    with a single value of type boolean.
softFault describes DCPS with a single value of type boolean.
softFail describes DCPS with a single value of type boolean.
lowRangeLimit describes DCPS
    with a single value of type decimal.
highRangeLimit describes DCPS
    with a single value of type decimal.
channels describes DCPS with values of type TwoChannels.

TwoChannels is a type of Channel, must be one of {Ch1, Ch2}.
Px is a type of DCPS. //Pressure Sensor

Channel is a class.
rawValue describes Channel
    with a single value of type UnittedQuantity.
convertedValue describes Channel
    with a single value of type decimal.
filteredValue describes Channel
    with a single value of type decimal.
hardFault describes Channel with a single value of type boolean.
hardFail describes Channel with a single value of type boolean.

rawValue of Channel only has values of type Signal_mV .

// Decompositions
SecondOrderTustinFilter is a class.
inp describes SecondOrderTustinFilter
    with a single value of type decimal.
_value describes SecondOrderTustinFilter
    with a single value of type decimal.

WeightedAverageTwoValue is a class.
inpA describes WeightedAverageTwoValue
    with a single value of type decimal.
inpB describes WeightedAverageTwoValue
    with a single value of type decimal.
_value describes WeightedAverageTwoValue
     with a single value of type decimal.
```

```
// Interface Definitions
COMPONENT_TEMPLATE_IntDef1
   is a INTERFACE_DEFINITION,
  with reference_class Channel,
  with reference_property rawValue,
  with functional_min 0.0 ,
  with functional_max 120.0 ,
  with physical_min 0.0 ,
  with physical_max 120.0 ,
  with tolerance 0.05 ,
  with resolution 0.01 .

COMPONENT_TEMPLATE_IntDef2
    is a INTERFACE_DEFINITION,
  with reference_class Channel,
  with reference_property convertedValue,
  with functional_min 0.0 ,
  with functional_max 500.0 ,
  with physical_min 0.0 ,
  with physical_max 500.0,
  with tolerance 0.05 ,
  with resolution 0.01 .

COMPONENT_TEMPLATE_IntDef3
   is a INTERFACE_DEFINITION,
  with reference_class Channel,
  with reference_property filteredValue,
  with functional_min 0.0 ,
  with functional_max 500.0 ,
  with physical_min 0.0 ,
  with physical_max 500.0 ,
  with tolerance 0.05 ,
  with resolution 0.01 .

COMPONENT_TEMPLATE_IntDef4
   is a INTERFACE_DEFINITION,
  with reference_class Px,
  with reference_property selectedValue,
  with functional_min 0.0 ,
  with functional_max 500.0 ,
  with physical_min 0.0 ,
  with physical_max 500.0 ,
  with tolerance 0.05 ,
  with resolution 0.01 .

COMPONENT_TEMPLATE_IntDef5
    is a INTERFACE_DEFINITION,
  with reference_class Engine,
  with reference_property engineSpeedPercentage,
  with functional_min 0.0 ,
  with functional_max 100.0 ,
  with physical_min 0.0 ,
  with physical_max 100.0 ,
  with tolerance 0.05 ,
  with resolution 0.01 .
```

We have 2 requirement files. In the first one below, the Context is used so that all the requirements in the file are grounded to the channels of *Px* which are *Ch1* and *Ch2* as defined in the ontology. This ability to do the grounding means that a requirements engineer needs to write fewer requirements than without grounding.

```
uri "http://sadl.org/PressureSensorReq" alias PSR.

import "http://sadl.org/PressureSensor".

Context: Px with channels TwoChannels.

// for Px channels : rawValue -> convertedValue -> filteredValue

Equation Conversion(decimal mult, decimal raw-input,
                    decimal adder) returns decimal:
mult * raw-input + adder.

// since "value" is a keyword, we use it as a non-keyword by
// escaping it with a preceding "^"
Requirement ConvertedValue:
SYSTEM shall set convertedValue of channels of Px to
Conversion(4.5, ^value of rawValue of channels of Px, -13.2).

Requirement FilteredValue:
SYSTEM shall set filteredValue of channels of Px
                    to _value of filteredSignal
where filteredSignal is (a SecondOrderTustinFilter
with description "2nd order Tustin with K=1, D=0.5, N=0.5"
with inp (convertedValue of channels of Px) ).


// for Px channels : hard fault -> hard fail (persisted hard fault)

Requirement HardFault:
SYSTEM shall set hardFault of channels of Px to true
only when convertedValue of channels of Px
                < lowRangeLimit of Px
        or convertedValue of channels of Px
                > highRangeLimit of Px.

Requirement HardFail:
SYSTEM shall set hardFail of channels of Px to true
only when hardFault of channels of Px has been true
          for the past 0.5 seconds.
```

The second requirements file is as follows.

```
uri "http://sadl.org/PressureSensorReq2" alias PSR2.

import "http://sadl.org/PressureSensor".

// for Px : set range limits

Requirement LowRangeLimit1:
SYSTEM shall set lowRangeLimit of Px to 1.0
when engineSpeedPercentage <= 30.0 .

Requirement LowRangeLimit2:
SYSTEM shall set lowRangeLimit of Px to 5.0
when engineSpeedPercentage > 30.0 .

Requirement HighRangeLimit:
SYSTEM shall set highRangeLimit of Px to 100.0 .

// for Px : soft fault -> soft fail (persisted soft fault)

Requirement SoftFault1:
SYSTEM shall set softFault of Px to true
when abs(convertedValue of Ch1 - convertedValue of Ch2)
        > 15.0
    and statusOk of Px is true.

Requirement SoftFault2:
SYSTEM shall set softFault of Px to false
when abs(convertedValue of Ch1 - convertedValue of Ch2)
        <= 15.0
    and statusOk of Px is true.

Requirement SoftFault3:
SYSTEM shall set softFault of Px to false
when statusOk of Px is false.

Requirement SoftFail:
SYSTEM shall set softFail of Px to true
only when softFault of Px has been true for the past 0.2 seconds.

Requirement PersistentBad:
SYSTEM shall set persistentBad of Px to true
only when hardFail of Ch1 is true
     and hardFail of Ch2 is true
     and softFail of Px is true.

Requirement SelectedValue:
SYSTEM shall set selectedValue of Px as follows:
{[hardFault of Ch1, hardFault of Ch2, persistentBad of Px,
                                        selectedValue of Px],
[false,  false,  --,     _value of
                         standardWeightedAvgPressureSensor ],
[false,  true,   --,     filteredValue of Ch1 ],
[true,   false,  --,     filteredValue of Ch2 ],
[true,   true,   false,  previous selectedValue of Px ],
[true,   true,   true,   14.0 ]
}
where standardWeightedAvgPressureSensor
        is (a WeightedAverageTwoValue
            with description "Standard Weighted Average"
            with inpA (filteredValue of Ch1)
            with inpB (filteredValue of Ch2)
           ).
```