University of Twente

Master Thesis

Using Requirement Templates to Automate Requirements Formalization

by

Tanja de Jong

August 2017

Abstract

Proving correctness of safety-critical software systems is potentially as important as the development of these systems. Verification tools can be used to automatically analyze a system, but this requires requirements to be specified in a suitable formalism. Currently, requirements are often separately specified in a natural language and subsequently formalized. This is time and effort intensive and error-prone, mostly because the natural language requirements might be ambiguous.

In this research, we present a framework (FRET) that allows users to specify unambiguous requirements in natural language, by using requirement templates that capture patterns between requirements. Furthermore, we discuss its functionality to automatically formalize these requirements into predetermined formalisms.

Through two case studies using ten realistic requirement specifications, we have evaluated this framework in comparison to the current situation of requirements specification and formalization. We found that FRET is currently capable of specifying all of the provided requirements and formalizing 94% of the requirements that could be manually formalized. Furthermore, FRET allows users to specify and formalize requirements in less time and with less effort and specification and formalization in FRET is less error-prone due to the effect of requirement templates on ambiguity, when comparing it to the current situation of requirements specification and formalization. Finally, we have established that two different formalizations of a default template are in fact equivalent to each other. Thus, we are capable of determining whether a template is consistent with regard to its set of formalizations.

Preface

The research described in this report has been carried out in behalf of Stinger Ghaffarian Technologies, Inc. at NASA Ames Research Center. This work is a collaboration with the FRET team at NASA ARC's Intelligent Systems Division. I would like to thank the entire FRET team, my supervisors and anyone else that participated in this research for their contributions.

Contents

List of Figures

List of Tables

Listings

Part I

Foundation

Chapter 1

Introduction

A safety-critical system is a system whose failure may result in death or serious injury to people or loss of or severe damage to property. Therefore, proving correctness of safety-critical software systems is potentially as important as the development of these systems. To establish correctness, one must first state what is expected of the system. In other words, the software requirements must be specified. Verification tools, such as model checkers and theorem provers, can be used to analyze these requirements. However, the specification of requirements and analysis of a system with regard to these requirements has proven to be surprisingly difficult for a number of reasons. First of all, the requirements must be unambiguous. Second, to establish a reliable degree of correctness of a software system, sufficient requirements must be identified. Finally, in order to use automated verification tools, the requirements must be specified in a notation that can be understood by these tools.

Two main approaches can be identified during the phase of requirements elicitation. The first approach is to write requirements in a formal language. This is the most convenient approach, if the aim is to later verify the software through an analysis tool that is based on that specific language. However, these formal languages are often too difficult to use and understand and thus require a lot of background knowledge. For that reason, this approach is not widely used.

The other, more popular, approach is to write requirements in natural language or an ad-hoc graphical notation. Unfortunately, this can result in ambiguous requirement specifications, since there is no unique mapping from natural language to its meaning as perceived by users. Different people involved in the software development process may interpret them differently, which can lead to a system that does not match its intended purpose. Furthermore, natural language requirements can not be directly used by software analysis tools, so an additional translation step is required.

This report discusses another approach in which requirements can be easily specified and formalized, without requiring a deep understanding of formal languages.

1.1 Objectives

This research aims to address the aforementioned issues in order to support requirement elicitation and verification. It combines the ease of requirement specification of approach two (specifying requirements in natural language or ad-hoc graphical notation) with the ease of automated model checking of approach one (specifying requirements in a suitable formal language). The goal is to provide a highly configurable framework for the development and analysis of unambiguous requirements. As a basis for this research, the FRET (**F**ormal **R**equirements **E**licitation **T**ool) project that is under development at the Robust Software Engineering group [1] at NASA Ames Research Center is used and extended.

The FRET project aims at providing a requirements development environment with the following features:

- An environment for the specification of unambiguous requirements. The requirements specification should make use of requirement templates to make it easier to specify unambiguous requirements. This includes the development of a metalanguage for the specification of requirements.
- Automated formalization and connection to several analysis tools. This feature should take advantage of the fact that requirements are based on templates and translate natural language requirements into selected formal formats, that can be used by certain verification tools.
- High configurability. This should allow for a wide range of requirements to be specified in FRET, while providing enough freedom for customization to make it more intuitive for use in particular projects.

The main objective of this research is as follows:

> *To (1) develop and (2) evaluate a framework for the unambiguous specification and automated formalization of requirements.*

For the development of the framework, the following questions must be answered:

(1a) **How can requirement templates be used to specify unambiguous requirements without requiring significant background knowledge of any formalism?**

(1b) **How can unambiguous requirements be formalized automatically into a chosen formalism?**

In order to evaluate the framework, we will consider the following attributes:

(2a) **Performance**
FRET should be able to support a wide range of requirements. The performance of the framework can thus be measured as the ratio of requirements that can be specified and formalized by using this approach.

(2b) **Usability**
The purpose of this research is to make it easier to specify and formalize unambiguous requirements correctly. Therefore, an important factor is the usability of the developed approach in comparison to alternative approaches, such as the elicitation of requirements in ad-hoc notations and the subsequent manual formalization. The usability is quantified using three metrics: time and effort required to specify and formalize requirements, the error-proneness of the specified and formalized requirements, and the ease of use of the framework.

(2c) **Consistency**
It is not feasible for this research to establish the correctness of any formalization of a natural language requirement, since this would require natural language processing, which is outside of the scope. However, some degree of correctness can be established in a number of ways. Since the goal is to formalize requirements into multiple formal notations, we can establish whether those different notations are equivalent to each other. This will determine whether formalization of a requirement in FRET is *consistent* between different target formats. Thus, in this context, consistent means that distinct formalizations of a single template are equivalent to each other.

1.2 Approach

The purpose of this research is to adapt and extend the FRET framework for the specification and automated formalization of requirements, in order to improve the process of system verification.

To enable the evaluation of FRET at the end of the research, we will first analyze the current situation of requirements specification and formalization. A set of safety-critical requirement specifications was provided by Lockheed Martin, a global aerospace, defense, security, and advanced technologies company. We will analyze these requirements and perform the first case study, in which these natural language, and possibly ambiguous, requirements are manually formalized for analysis of their corresponding systems by CoCoSim [2], an analysis tool that is based on Satisfiability Modulo Theories for model checking.

Next, we adapt and extend the FRET framework to answer research questions (1a) and (1b). The requirements specification component is developed based on requirement patterns that inspire templates. For the requirements formalization component, we use these templates to transform the natural language requirements into a formal language. To demonstrate this, we define a formalization into a format that can be analyzed by CoCoSim. We also define a formalization into Metric Temporal Logic, which is an extension of Linear Temporal Logic (LTL).

Finally, a second case study will be performed to analyze the specification and formalization of the Lockheed Martin requirements using the FRET framework. This will lead to an evaluation of FRET's performance (2a). The results of the first and second case study will be compared to obtain an indication of FRET's usability, compared to the manual specification and formalization of requirements (2b). Finally, the formalizations into CoCoSim and MTL will be checked for equivalence, to determine whether FRET's formalizations are consistent (2c). This requires that both the CoCoSim and MTL formalizations are transformed into an intermediate format, for which we use Lustre, a synchronous dataflow programming language, because it is already used by CoCoSim.

1.3 Contributions

As mentioned before, FRET was already under development before this research started. Much time and effort had already been put into research regarding templates and the FRET team had already experimented with quite a few of them. This had already resulted in a single basic template that covers a broad range of requirements and the possibility of defining domain patterns that customize existing templates (without requiring new formalization), with a focus on configurability in order to adapt to different projects and users. Also, much work had already been put into supporting LTL formalization for the basic template.

A considerable amount of engineering was already in place, so that the tool previously enabled the definition of templates and the specification of requirements based on templates.

Clear contributions of this research are the formalization for and connection of FRET to CoCoSim as well as two case studies: one in which we specify requirements in natural language and then manually formalize them, and one in which we use FRET to specify and automatically formalize requirements. This provides insight in the performance and usability of FRET.

Finally, a redesign of FRET was performed, which leads to the version of FRET that is described in the following chapters.

1.4 Structure

First, Chapter 2 reviews existing work on requirements engineering, specifically on requirements patterns and templates, SMT-based model checking, CoCoSim, Metric Temporal Logic and Lustre. Next, Chapter 3 provides an evaluation of the current situation of formalizing natural language requirements.

After this, the development phase is discussed in Chapter 4. This starts with an overview of the FRET architecture, after which the we discuss how patterns can be used to develop templates that help specify unambiguous requirements. Then, we discuss how these patterns can be used to formalize these requirements and explain how FRET provides high configurability that further sets it apart from existing work.

Finally, FRET is evaluated. Chapter 5 discusses performance, Chapter 6 discusses usability, and Chapter 7 discusses consistency. Chapter 8 gives the conclusions that can be drawn based on the results of the research and Chapter 9 discusses ideas for future work.

Chapter 2

Literature review

2.1 Software safety

Parnas et al. [3] provide an evaluation of safety-critical software. They discuss a number of techniques that are commonly used in the evaluation of a software system, such as testing, reviewing and design principles, and the questions that these techniques pose.

Leveson [4] provides a survey that explains why there is a problem regarding software safety, what the problem is, and what is known about how to solve it. She states that the goal of system safety is to "design an acceptable safety level into the system before actual production or operation". However, the degree of complexity in software makes it impossible to demonstrate that the design of software is completely correct in order to fully eliminate faults. Furthermore, she states that a fundamental misunderstanding of requirements may actually be the most important source of software problems with respect to safety.

Young and Leveson [5] discuss how significant progress is lacking in the software safety area. To the contrary: traditional safety analysis techniques are becoming less effective due to changes in engineering that cause systems to become increasingly complex. They suggest an approach in which systems theory is used to achieve system safety and security. Leveson [6] also considers systems theory in engineering for system safety, through Rasmussen's idea. Rasmussen's original model has a formal, mathematical foundation built on systems theory. Leveson's paper describes three applications of Rasmussen's idea to systems engineering practice in order to improve system safety.

Finally, Stringfellow et al. [7] describe yet another process for safety-driven development. It combines three techniques: from organizing the development of a system in a hierarchical structure, to a framework upon which to base safety engineering techniques, to hazard analysis. Each of these techniques benefits from a clearly defined requirements specification, because this is what a system will be evaluated against. Requirements engineering should serve as the basis for determining system safety.

2.1.1 Requirements engineering

The entire process of specifying system requirements through to the analysis of a system with regard to these requirements is called requirements engineering [8]. The gathering of requirements is called requirements elicitation. This should result in a structured document that specifies the services that a system is expected to provide. This requirements document should serve as a basis for all other system development activities, including the validation and verification of the system [9].

Requirements engineering contains a lot of different aspects and can be viewed from business, behavioural and software engineering perspectives. According to Loucopoulos and Karakostas [10], requirements engineering can be considered a combination of three concurrent and interacting processes: eliciting knowledge related to a problem domain, ensuring the validity of such knowledge and specifying the problem in a formal way [10]. They mostly focus on requirements elicitation techniques, whereas our research focuses on how to unambiguously specify and formalize the elicitated requirements.

Pohl presents approved techniques for various aspects of the requirements engineering process, among which the documentation and validation of requirements [9]. The need for these techniques indicates the complexity of unambiguously specifying correct requirements.

In 2000, Nuseibeh and Easterbrook [11] provided a roadmap for requirements engineering. They provide an overview of current research in a number of activities that they consider core RE activities, among which the modeling and analysis of requirements and the communication of requirements. Thus, for requirements engineering, it is important that requirements are both easy to write and read and easy to analyze. Our research focuses on the combination of these two aspects. One attempt to achieve readability has been the development of a variety of documentation standards that provide guidelines for structuring requirements documents, but some authors, such as Kovitz [12], argue that templates cannot in themselves provide a general structuring mechanism for requirements. Rather, the structure has to be developed for the particular context in hand.

A more recent overview is given by Cheng and Atlee [13] in 2007. They begin by giving a number of reasons why RE is inherently difficult. One of these reasons is again that the resulting artifacts must be understood and usable by both domain experts and stakeholders with less domain background. So, requirements notations must maintain a delicate balance between producing descriptions that are easily understandable and producing technical documents that are precise enough for developers. In their research, they mainly target their distinction towards the 'technical people with background knowledge' versus the 'non-computing audience'. However, since we focus on the connection between requirement specifications and software verification, we can also consider the people that are responsible for the requirements analysis and software verification to require precise and technical documents, even though they might not have specific domain knowledge regarding the context of the system. [13] agrees with [11] that requirements modeling, analysis and validation as well as system verification are core RE tasks. They identify two categories for requirements analysis, namely analyses that look for well-formedness errors (ambiguity, inconsistency, incompleteness) and analyses that look for anomalies. Our research targets the first type of analysis. Among the RE research hotspots that [13] identifies is 'Methodologies, Patterns, and Tools'. They state that most research projects focus on a single RE problem, such as elicitation or traceability, with little knowledge of how to combine techniques effectively. They suggest that research should focus on how to integrate different aspects of RE. Our research focuses on creating a single approach for requirements documentation, analysis, and formalization and software verification.

2.1.2 Ambiguity

However, informal requirements documents are often ambiguous, incomplete, and even inconsistent. Because requirements serve as the basis for all other development activities, errors in the requirements documents can propagate through all of these phases. Gervasi and Zowghi [14] investigate the nature of ambiguity and provide a classification of several types of ambiguities, to provide a better understanding to help in the analysis of ambiguity and the design of more effective methods to detect, mark and handle ambiguity. They give the following definition of ambiguity: "the phenomenon by which multiple distinct meanings can be assigned to the same requirement". As a reference frame, they use that of the classical denotational approach, where semantics is given by a *function* mapping from a *source domain* to a *target domain*. This is the definition of ambiguity that we will use throughout this report.

De Bruijn and Dekkers [15] provide a case study of ambiguity in natural language requirements, in order to determine the effect of ambiguity on project failure. This case study shows that highly ambiguous requirements do not necessarily cause the major issues of software projects, but it requires a significant investment from the development team to clarify the requirements and cope with the high level of ambiguity.

According to Zowghi and Coulin [16], the large degree of error that requirements elicitation is subject to is mainly influenced by communication problems. Communication problems cause and are in turn caused by ambiguous requirements specifications. Berry and Kamsties [17] identify the problem of ambiguity in natural language requirements. According to an online survey, 79% of requirements specification documents are written in common natural language, 16% are written in structured natural language and only 5% are written in formalized language [18]. Only if a requirement is written in a formalized language, the semantics are fixed and thus ambiguity can be prevented. On the other hand, common natural language greatly decreases the chances of creating an unambiguous requirements specification.

Gause [19] identifies unrecognized disambiguation as one of the most important sources of requirements failure. Unrecognized disambiguation occurs when a reader of a text takes the first meaning that comes to mind as the only meaning of the text, which may be entirely wrong. It is of major importance that this problem is dealt with sufficiently, but this is difficult because it happens unconsciously. Therefore, Kamsties [20] has analyzed ambiguity in requirements elicitation. The results show that formalization during further software development activities does not solve ambiguity problems, but they indicate that a previous inspection using a tailored reading technique can detect most ambiguities that slip through formalization. Therefore, he suggests to use a combination of a reading inspection of the requirement with subsequent formalization. In [21], Kamsties and Peach build on this research by showing how to detect ambiguities in natural language requirements using a checklist.

2.1.3 Requirements formalization

According to De Groot and Hooman [22], formalizing the requirements of a system helps to discover the ambiguities and inconsistencies in an informal specification. Their experiments show that the simple act of formalizing the requirements helps to discover many of the problems of an informal specification.

Fraser et al. [23] discuss the differences between informal and formal requirements specification languages and suggest two approaches to bridge the gap between informal and formal requirements specifications. Their first approach uses an informal model of a system to guide an analyst's understanding of the system and the development of a formal specification. Thus, this is a fully manual formalization technique. Their second approach proposes a rule-based method for generating formal specifications from a set of informal specifications. This rule-based approach can potentially form the basis for an automated formalization technique.

Bubenko et al. [24] further discuss how the major problem with bridging the gap between informal and formal requirements specification is how ill-defined problem and application situations are. They outline an approach that uses models that reflect the enterprise and its objectives. Their research focuses on gradually achieving a formal specification through a number of iterations by increasing a developer's understanding of a system's surroundings, instead of attempting to immediately prevent any ambiguity in a specification.

Leveson et al. [25] describe an approach for writing formal requirements specifications that specifically target process-control systems. Their research demonstrates the practicality of writing a formal requirements specification for a complex system and the feasibility of building a formal model of a system using a specification language that is readable by non-experts. Thus, it focuses on creating requirement specifications that are easy to understand, but unambiguous at the same time.

2.1.4 Requirements templates

According to Dwyer et al. [26], the general structure of requirements is often captured by a number of patterns. By exploiting the similarities between requirements, a pattern-based approach can be used for both the elicitation and formalization of requirements. Requirement *templates* can be defined, that are directly associated with appropriate formal semantics, and end users only need to select a template and instantiate it with their specific requirements. Thus, basically it uses a rule-based approach as in the second approach of [23], so that templates are mapped to their formal semantics. In their research, eight basic patterns have been defined:

- **Absence:** a given state/event does not occur within a scope.
- **Existence:** a given state/event must occur within a scope.
- **Bounded existence:** a given state/event must occur k times within a scope.
- **Universality:** a given state/event occurs throughout a scope.
- **Precedence:** a state/event P must always be preceded by a state/event Q within a scope.
- **Response:** a state/event P must always be followed by a state/event Q within a scope.

Pattern	**Total**	**Percentage**
Absence	85	15%
Universality	119	21%
Existence	26	5%
Bound existence	1	0%
Response	245	44%
Precedence	26	5%
Chain precedence	1	0%
Chain response	8	1%
No pattern	44	8%
Total	555	100%

TABLE 2.1: Number of instantiations per pattern

- **Chain Precedence:** A sequence of states/events $P_1, ..., P_n$ must always be preceded by a sequence of states/event $Q_1, ..., Q_m$.
- **Chain Response:** A sequence of states/events $P_1, ..., P_n$ must always be followed by a sequence of states/event $Q_1, ..., Q_m$.

They conducted a survey with 555 property specifications to determine how many of them are instances of the defined patterns. The total number of specifications that matches each pattern is listed in Table 2.1. It shows that a large majority of the property specifications match the response pattern.

Smith et al. [27] discuss how requirement templates can be used during the process of requirements specification that make use of requirement patterns. A requirement can be specified by instantiating a matching template. This can reduce ambiguity, because a user only needs to deal with subtle details when the template is being specified, after which the requirements can easily be specified unambiguously. They applied their work in a tool called Propel and focus on discovering how to make property templates more usable for users who are not very familiar with specification formalisms. This consists of determining the various ways in which an informal requirement might be interpreted and helping the user to remove any ambiguity.

Propel can represent templates in *disciplined natural language* (DNL) or as finite-state automata (FSA). These are used as equivalent alternatives, so we will explain the basics of Propel using DNL.

A DNL template consists of a *Core phrase* that captures the basic meaning of a requirement and optional additional phrases. A Core phrase can be selected from a list of equivalent phrases. The user can select the one that suits the domain of the requirement best. Furthermore, every phrase can contain parameters that are used to instantiate the template. For example, the DNL template that was defined in [27] for the *Response*

Core phrase	**Pre-arity** occurrences of action **Immediacy** lead to **Post-arity** occurrences of response.
	Pre-arity occurrences of response **Immediacy** respond to **Post-arity** occurrences of action.
Nullity phrase	Action may occur zero times.
	Action must occur at least once.
Precedency phrase	Response cannot occur before the first action occurs
	Response may occur before the first action occurs.
Repetition phrase	The behavior above is repeatable.
	The behavior above is not repeatable.

Table 2.2: Partial DNL template for response

pattern of [26] is partially given in Table 2.2. The words shown in boldface are the fields that must be instantiated by choosing any of a predetermined list of options.

Propel was later extended with *Question Trees*. A Question Tree asks a user a series of questions about subtle aspects of a requirement, to help guide users through the instantiation of requirements. In a typical requirements specification process, the user is fully responsible for recognizing all of these subtle aspects without any help. The use of Question Trees in Propel is discussed in [28].

2.2 CoCoSim

A software system can often be represented as a *state transition system*. Such a system can be seen as a model, both in the sense that it is an abstraction of the real system and that it is mathematical. The functional properties of such a system can be expressed as *temporal* properties for a suitable model of that system in a suitable temporal logic. Given a model and corresponding correctness properties, model checking refers to exhaustively and automatically verifying the properties for that model.

Model checking can be done using Satisfiability Modulo Theories (SMT) [29] techniques. SMT is a *decision problem* for logical first order formulas with respect to combinations of background theories. An SMT solver is a tool for deciding the satisfiability of formulas in these theories.

CoCoSim [2] is an automated analysis and code generation framework for Simulink and Stateflow models that is interfaced with several external SMT-based solvers. It is partially being developed at NASA Ames Research Center, so it was chosen for this research in order to also investigate how it could be combined with FRET to improve V&V at NASA. For this research, release v0.1 is used.

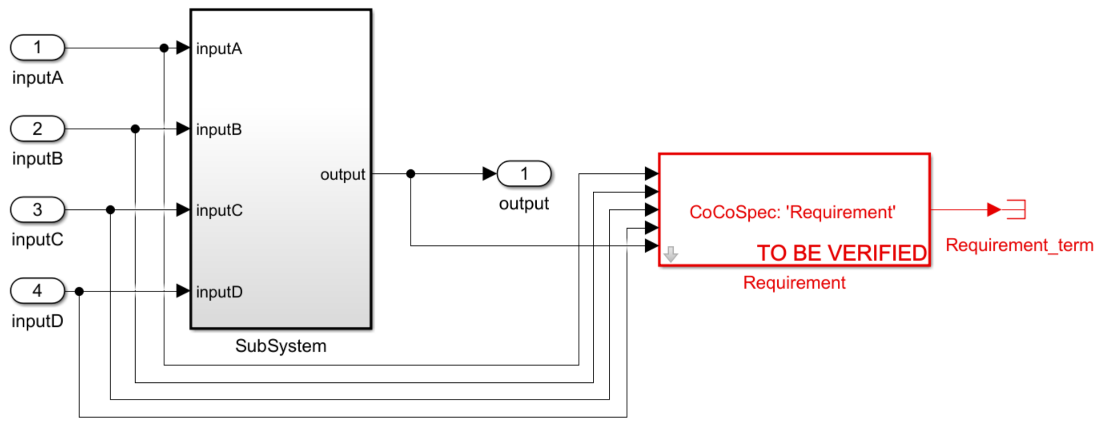


FIGURE 2.1: A Simulink model with CoCoSim property

Simulink [30] is graphical programming environment for dynamic systems that is integrated in Matlab. Stateflow [31] is an optional environment within Simulink that can be used to model reactive systems specifically, using state machines and flow charts. During this research, we have only used Simulink models that do not contain any Stateflow aspects.

A Simulink model consists of a set of inputs and outputs, and blocks between them that perform the program logic. We consider such models to be purely discrete. Thus, at every fixed step, computations occur only for the next time step; continuous states are not computed.

A safety requirement is represented as a boolean function on the relationship between the input and output blocks. A CoCoSim safety requirement can be added to a model as a subsystem block, as shown in Figure 2.1. This figure shows a Simulink model with four inputs, `inputA`, `inputB`, `inputC`, and `inputD`, and a single output, `output`. The program logic is grouped inside the `Subsystem` block, that takes the inputs and produces the output. The details of the program logic are thus not shown in this figure. The CoCoSim requirement is the red block shown to the right of the model. It takes all the inputs and the output of the model. The output port of the requirement is capped by connecting it to a terminator block, because there is no use in connecting this output to the original model; it is used directly by CoCoSim. The CoCoSim subsystem represents the given requirement, so if we then run one of the solvers that CoCoSim is interfaced with, it will verify through model checking whether the Simulink model satisfies the requirement or not. The CoCoSim requirement subsystem will be classified as `SAFE` if and only if this is the case.

CoCoSim is currently interfaced with three external solvers: Zustre [32], Kind2, and JKind [33]. These are all SMT-based model checkers for Lustre programs. Zustre

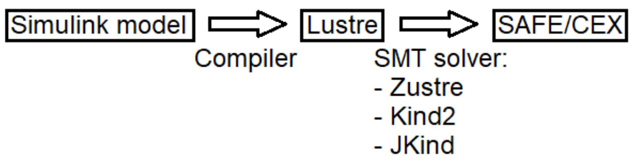


FIGURE 2.2: CoCoSim's analysis workflow

was developed by the CoCoSim team, while the others were developed separately from CoCoSim.

If one of the solvers is executed, CoCoSim compiles the model into Lustre code as its intermediate language and then runs the selected solver on the generated Lustre code, as illustrated in Figure 2.2. The solver checks the system model against all the CoCoSim requirements that were added to the model and results in either a safety declaration or a counterexample for each CoCoSim requirement.

When adding a requirement, the user must select the path to the subsystem that the requirement should be added to, the outputs that are relevant to the requirement and the name of the requirement. CoCoSim will then add a masked observer to the selected subsystem, which means that the observer is encapsulated in its own block diagram within the model. It is by default connected to all the inputs and the selected outputs of that subsystem. Next, it is the user's task to connect the inputs and outputs within the observer through Simulink blocks in such a way that it represents the corresponding requirement. An example of the internals of a CoCoSim requirement is given in Figure 2.3. It represents the following requirement:

> "If `inputA` is false and `inputB`, `inputC`, and `inputD` are all true, then `output` must be true."

After the user is finished modeling the requirement, one of the supported SMT solvers can be executed.

Finally, the requirement subsystem is updated to reflect the results of the verification attempt: it becomes green if the property has been declared safe, red if a counterexample was found, or another color if a problem occurred. Additionally, the 'TO BE VERIFIED' text is replaced by a description of the results.

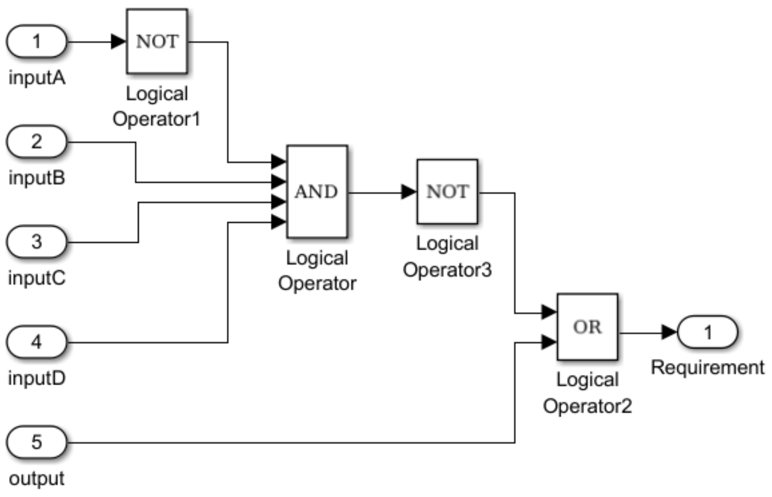


FIGURE 2.3: Internals of a CoCoSim property

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
#Requirements	4	13	4	10		4	4		2	8
#Blocks	479	279	45	291	355	699	75	355	141	61

TABLE 2.3: Overview of LM challenges

2.3 Lockheed Martin challenges

As a general verification challenge, a set of ten requirement specifications was made available by Lockheed Martin. We chose to use these as a case study to evaluate FRET and compare it to the current situation of requirement specification and formalization.

These requirement specifications will be referred to as the LM challenges from this point forward. Each challenge contains a Simulink model for a realistic aerospace (sub)system, a specification document with a description of the model, and a list of requirements in natural language that the model should comply with.

Table 3.1 gives an overview of the number of requirements per challenge and the number of blocks in each Simulink model.

2.4 Metric Temporal Logic

Linear Temporal Logic [34] (LTL) is a formal logic for specifying and verifying properties of reactive systems. In LTL, it is possible to encode properties about the future of paths.

An LTL formula describes for which sequences it is true. If all of a system's possible sequences are in a property's set, the system satisfies that property.

LTL's formulas can contain the usual logical connectives, as well as temporal operators:

X ϕ (ne**x**t) ϕ will hold in the next state

ϕ **U** ψ (**u**ntil) ϕ holds some time in the future, and ψ holds at least up to the first ϕ

G ϕ (**g**lobally) From now on, ϕ holds

F ϕ (**f**uture) Eventually (in the future) ϕ will hold

ϕ **R** ψ (**r**elease) Either ϕ holds indefinitely, or ϕ is true until and including the point when ψ is first true

W (**w**eakly until) Either ϕ holds indefinitely, or ϕ is true until ψ is true

Metric Temporal Logic [35] (MTL) extends Linear Temporal Logic by constraining the temporal operators by (bounded or unbounded) intervals of the real numbers. Because we focus on discrete real-time systems, these real number intervals directly correspond to the time steps in which real-time systems perform their computations.

For example, the formula $\mathbf{F}_{[1,4]}\phi$ means that ϕ will become true within 1 to 4 time units from now. If we have a requirement that states that, whenever ϕ occurs, ψ must occur immediately, we can formalize this as $\mathbf{G}\phi => F[1,1]\psi$. In other words:

> Globally, if ϕ holds in some state, eventually within 1 and 1 time steps from that state (thus immediately), ψ must hold.

Thus, in the context of this research, we only use intervals with the **F** operator.

2.5 Lustre

Lustre [36] [37] is a synchronous data flow language for real-time systems. A data flow architecture [38] bases its execution solely on the availability of input arguments, so that the order of instruction execution is unpredictable. This contrasts the traditional von Neumann [39] or control flow architecture, that uses a program counter and executes instructions in a specified order. A data flow program [40] implements the data flow architecture and is a directed graph where each node represents a function and each arc represents a signal path. If a data flow is synchronous, the number of data samples each node produces or consumes is specified a priori.

A Lustre program is a system of equations, in which each equation defines a variable. These variables are functions from time to their domain of values. In discrete systems, which is what we focus on, time is projected onto the set of natural numbers. Thus, variables are infinite sequences of values.

As an example, if `X` is a variable and `E` is an expression, the equation `X=E` defines `X` to be the sequence

$$(x_0 = e_0, x_1 = e_1, ..., x_n = e_n, ...),$$

where $(e_0, e_1, ..., e_n, ...)$ is the sequence of values of the expression `E`.

Expressions consist of variables, constants, and operators. The usual arithmetic, boolean, and conditional operators are called *data operators* and are extended to operate over sequences instead of single values. For instance, the expression

```
if X>Y then X else Y
```

is the sequence whose n-th term is the maximum of the n-th values of `X` and `Y`.

In addition to data operators, Lustre contains a number of non-standard operators, called *sequence operators*, which actually manipulate sequences. Among these sequence operators are `pre` and `->`. `pre` ("previous") is a memory operator that is used to keep track of the value of an expression from one cycle to the next. To initialize variables, the `->` ("followed by") operator is introduced. If `X` $= (x_0, x_1, ..., x_n, ...)$ and `Y` $= (y_0, y_1, ..., y_n, ...)$ are two variables of the same type, then

$$\texttt{X -> Y} = (x_0, y_1, y_2, ..., y_n, ...)$$

This means that `X -> Y` is equal to `Y` except at the first instant. As an example of use of these operators, the equation

$$\texttt{X} = 0 \texttt{ -> pre(X) + 1;}$$

defines `X` to be 0 initially, and its previous value incremented by 1 subsequently. So, `X` is the sequence of natural numbers.

A Lustre node is a Lustre subprogram. It receives input variables, computes output variables (and possibly local variables) through a system of equations. Listing 2.1 shows an example of a general counter implemented as a Lustre node.

```
1 node COUNT (init, incr: int; reset: bool)
2     returns (n: int);
3 let
4     n = init -> if reset then init else pre(n) + incr;
5 tel;
```

Listing 2.1: A counter in Lustre

`COUNT` is the name of the node. It expects three input variables: `init`, `incr`, and `reset`. Additionally, it returns one output variable: `n`. The output `n` is computed through an equation (lines 4 and 5). It is initially zero (line 4). Subsequently, if `reset` is true, `n` is set to `init` again. Otherwise, it is equal to its previous value plus the value with which it should be incremented. For example, it is possible to state:

$$\text{even} = \texttt{COUNT}(0, 2, \texttt{false});$$

This means that even is the sequence of even numbers.

Chapter 3

Evaluation of Current Situation

The current situation of requirement elicitation and formalization is evaluated through a case study. We attempted to formalize a set of requirements provided by Lockheed Martin, a global aerospace, defense, security and advanced technologies company [41], into a format that can be analyzed by CoCoSim. This chapter first provides some background information about the Lockheed Martin requirements and then gives an overview and evaluation of the results of the case study.

3.1 Lockheed Martin challenges

As a general verification challenge, a set of ten requirement specifications was made available by Lockheed Martin. We chose to use these as a case study to evaluate FRET and compare it to the current situation of requirement specification and formalization.

These requirement specifications will be referred to as the LM challenges from this point forward. Each challenge contains a Simulink model for a realistic aerospace (sub)system, a specification document with a description of the model, and a list of requirements in natural language that the model should comply with.

Table 3.1 gives an overview of the number of requirements per challenge and the number of blocks in each Simulink model. Appendix A contains a more detailed specification of the challenges.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
#Requirements	4	13	4	10		4	4		2	8
#Blocks	479	279	45	291	355	699	75	355	141	61

TABLE 3.1: Overview of LM challenges

		Requirements												
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Challenges	**0**	🟩	🟩	🟧	🟩									
	1	🟩	🟩	🟧	🟧	🟩	🟩	🟧	🟩	🟧	🟩	🟧	🟩	🟩
	2	🟩	🟩	🟩	🟧									
	3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩			
	4	🟥	🟥	🟥	🟥	🟥	🟥	🟥						
	5	🟩	🟩	🟩	🟩									
	6	🟥	🟥	🟥	🟥	🟥								
	7	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥		
	8	🟩	🟩											
	9	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥					

TABLE 3.2: Results of case study 1

3.2 Results

Table 3.2 shows the results of our attempt to formalize the LM challenges into the CoCoSim format. The green cells are requirements that we could immediately formalize. The orange cells are ambiguous requirements that required some changes before formalization was possible. These updated requirements are also given in Appendix A. The red cells are requirements that could not be formalized:

- Challenges 4 and 7: lack of specific domain knowledge. The requirements are not specified directly in terms of the model's inputs and outputs and we could not understand the meaning of the requirements. Thus, we could not formalize them.
- Challenges 6 and 9: library that contains blocks for matrix arithmetic is missing, so these models are incomplete. Furthermore, the requirements will also need to use blocks from this library.

The 'Discussion' sections of Appendix A discuss any problems that we encountered with the requirements in more detail.

Because the requirements in green were clearly specified, it was rather easy to formalize them. For example, the following requirement is part of challenge 2:

2. The Output (yout) shall be bounded by the provided Top and Bottom limits.

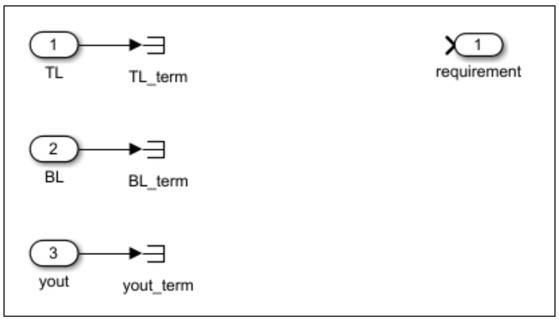


FIGURE 3.1: Requirement 2.2 as created by CoCoSim

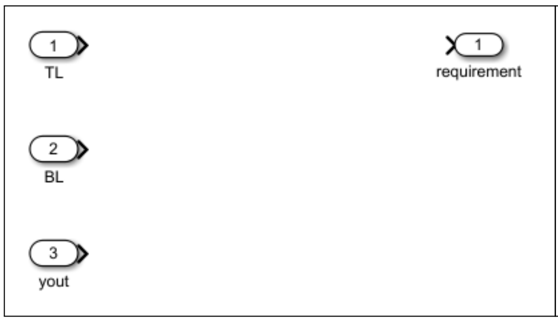
This means that $yout <= TL$ and $yout >= BL$ must always be true.

This is a very simple requirement, but it requires the following steps to create it in CoCoSim:

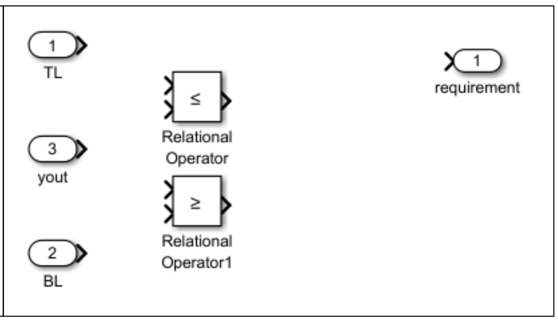
1. In the CoCoSim menu, select the option to add a new requirement.
2. Enter the subsystem to which the requirement should be added.
3. Enter a name for the requirement.
4. Enter which outputs the requirement will use (`yout`). CoCoSim will now create the requirement as shown in Figure 3.1.
5. Add/remove blocks and connect them to model the requirement as illustrated by Figure 3.2.
   (a) Remove terminator blocks that terminate `yout`, `TL`, and `BL`.
   (b) Add two relational operator blocks and set the operator parameters to $<=$ and $>=$.
   (c) Add a logical operator (the default is already set to 'and').
   (d) Connect all the blocks.

This may also require the rearranging of blocks and adding `GOTO` and `FROM` blocks, which allow to pass a signal from one block to another without actually connecting them, to provide a better overview and make it easier to create more complicated requirements.

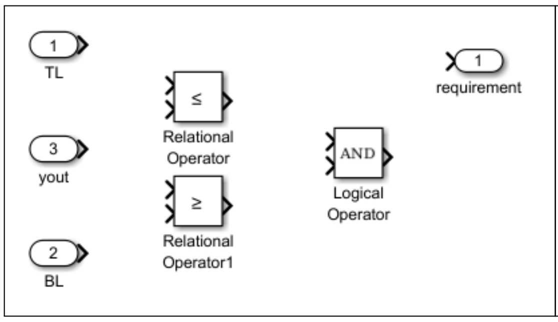
So even for the requirements in green, it still takes up a significant amount of time and effort to create and connect the Simulink blocks and correctly set their parameters. For the requirements in orange, additional time to analyze and rephrase the requirements was needed. In some cases, the ambiguity was not apparent and it was only after CoCoSim returned a counterexample that we realized some details were missing. A counterexample shows a list of values for the inputs and outputs that are possible in



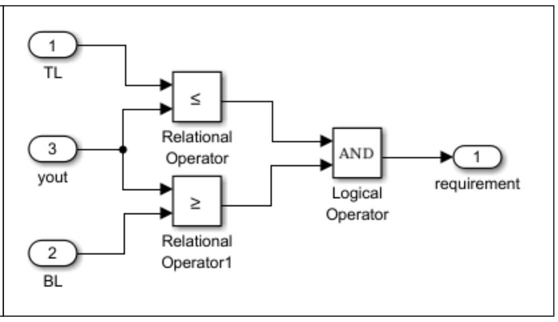
(A) Remove terminator blocks



(B) Add relational operator blocks



(C) Add logical operator block



(D) Connect blocks

FIGURE 3.2: Contents of the `deactivate` block

`inputA`	false
`inputB`	true
`inputC`	true
`inputD`	true
`output`	false

FIGURE 3.3: Possible counterexample

the given system model, but that violate a CoCoSim requirement. For example, for the requirement illustrated by Figure 2.3, a possible counterexample is given in Table 3.3.

Thus, it is easy to miss any errors that exist in a natural language specification when formalizing that specification, because unambiguities often go unnoticed.

For example, the following requirements are part of challenge 1:

2. The autopilot shall change states from TRANSITION to STANDBY when the pilot is in control.

3. The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.

At first sight, these requirements seem quite obvious. However, what happens if the autopilot is in state TRANSITION and the pilot is in control, the system is supported and the sensor data is good? Should the state change to STANDBY or to NOMINAL?

The description of the system indicates that the system should adhere to requirement 2, so we change 3 to:

> The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good <u>and the pilot is not in control</u>.

Considering the requirements in red, in some cases they were not sufficiently specified in terms of the model's inputs and outputs for non-experts in the domain to make sense of them. For example, requirement 1 of challenge 4 states:

4. NLGuidance shall always maintain the target on the port-side of the vehicle.

However, the model only contains inputs `Xtarg`, `Xv`, `Vy`, `r`, `Vt` and output `yout`. Now, how should we use these inputs and outputs to determine what the target and the vehicle are, and what the port-side of the vehicle is?

3.3 Conclusion

It was rather challenging to formalize these requirements, mainly due to ambiguous requirements and specifications that require a lot of domain knowledge in order to understand how they connect to the corresponding Simulink models.

Some requirements are missing details that probably seemed very obvious to the person who specified them, but can easily be overlooked by the person who formalizes these requirements without the entire system in mind, such as the requirements of challenge 1. When analyzing these requirements, we unconsciously starting thinking of them as patterns in order to resolve the ambiguities. For example, by mapping requirements 2 and 3 to the pattern 'The autopilot shall change states from TRANSITION to _ when standby is _, supported = _, good = _', we realized that they overlapped when standby, supported and good were all true.

Another problem is that much background knowledge can be required when formalizing previously specified requirements, which is the case for challenge 4. This cannot easily be solved without learning significantly more about the requirements' domain. However, if more focus was put on the link between the specification and the model, this could have been prevented.

We expect that using a more structured approach during requirements specification can prevent both of these issues.

Part II

Development

Chapter 4

Development

This chapter describes the state of the FRET project as of the end of the current research, which includes the contributions made by this research as described in Section 1.3.

A high-level architecture of the FRET framework is given in figure 4.1.

It shows that FRET consists of two major parts: the resources and the functional components. A *subject matter expert* can edit the resources. A subject matter expert should have extensive knowledge of the domain of a project and should be capable of configuring FRET to reflect this domain. This includes editing and creating templates, parsers and ontologies, which is discussed in more detail in chapter 4.3.

The second role involved with FRET is the formal methods expert. This person is responsible for creating the formalizations for the templates.

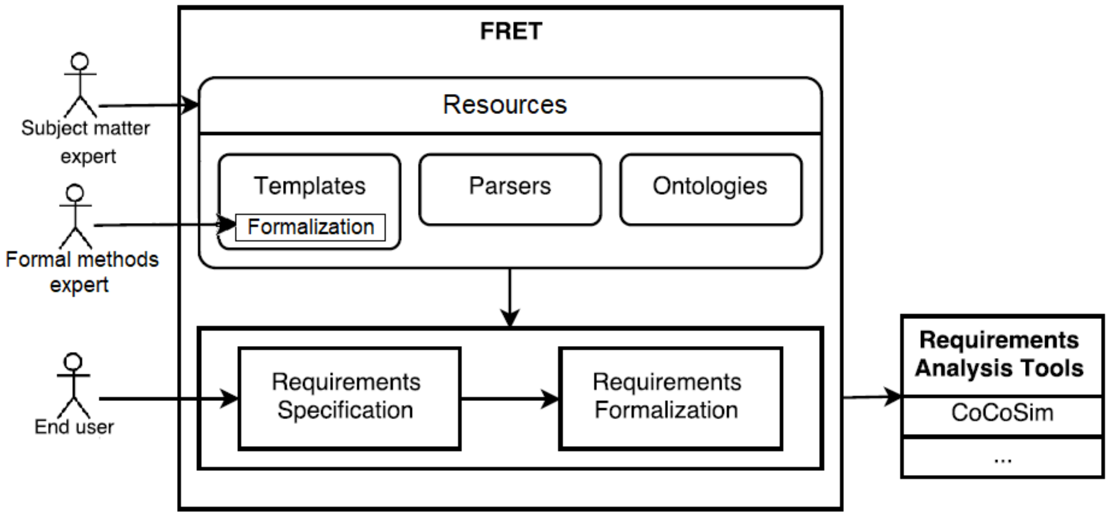


FIGURE 4.1: FRET architecture

The final role involved with FRET is the end user, who interacts with the functional part of FRET. End users are responsible for the specification of the requirements of a project.

The functional part of FRET consists of two components: the requirements specification component and the requirements formalization component. The requirements specification component allows end users to specify requirements by using FRET's resources and stores them. The requirements formalization component uses these stored requirements as well as the resources to formalize the requirements into a format that is supported by the chosen template. Through these formalizations, FRET can be interfaced with requirements analysis tools, such as CoCoSim, to automate the analysis of the specified requirements.

All of these components are discussed in more detail in the following sections.

4.1 Requirements Specification

FRET is built around the concept of requirement templates. This approach allows for unambiguous requirement specification and automated formalization.

As discussed in section 2, Smith et al. [27] define a DNL template as consisting of a Core phrase that captures the basic meaning of a requirement, with optional additional phrases through which users can answer specific questions about the requirement to remove ambiguities.

Because a large majority of property specifications match the response pattern of [27], an altered version of this pattern that better matches the type of safety-critical requirements that this research focuses on, is used as a basic template of FRET. The alterations are based on sets of available requirements that represent the safety-critical domain that this research targets. This template is called the *Conditional Event Logic* (CEL) template. In the template format of [27], the CEL template can be specified as in Table 4.1.

This chapter discusses how requirements can be specified in FRET and how templates and other features are used for this purpose.

4.1.1 Template format

An example of the CEL template in the FRET format is given in Table 4.2. In FRET, a template consists of the following basic structure:

Core phrase	Whenever **Trigger** and **InitialConditions**, then **PostConditions** must become true before **Time**.
Trigger phrase	If a new Trigger occurs, it releases the obligations set by the previous Trigger.
	If a new Trigger occurs, it does not release the obligations set by the previous Trigger.
Persistence phrase	The InitialConditions need to be true only at the time where the Trigger is true.
	The InitialConditions need to persist until the PostConditions are true.

Table 4.1: DNL template for Conditional Event Logic

Name	Conditional Event Logic	
Fields	Whenever	
	<Co[Trigger]:TR>	
	and	
	<Co[InitialConditions]:IC>	
	, then	
	<Co[PostConditions]:PC>	
	must become true within	
	<Co[Time]:TI>	
	.	
Questions	Q1: If a new Trigger occurs, does it release the obligations set by the previous Trigger?	`<Boolean>`
	Q2: Do the InitialConditions need to be true only at the time where the Trigger is true?	`<Boolean>`
Field types	TR	<Any>
		true
		<Parser:Condition>
	IC	<Any>
		true
		<Parser:Condition>
	PC	<Any>
		<Parser:Condition>
	TI	<Any>
		1 frames
		<Parser:Duration>

Table 4.2: FRET template for Conditional Event Logic

Name A descriptive name by which the template can be identified, such as 'Conditional Event Logic'.

Fields A list of all the fields of a template. The Core phrase is broken down into a list of static text and parameter fields in the same order as the sentence. A static field is simply static text. A parameter field represents a parameter in the template that should be further specified by a user upon instantiating the template with a requirement. For the CEL template, the Core phrase would be broken down into the following field list:

1. Whenever (static)
2. Trigger (parameter)
3. and (static)
4. InitialConditions (parameter)
5. , then (static)
6. PostConditions (parameter)
7. must become true before (static)
8. Time (parameter)
9. . (static)

We defined a metalanguage for the specification of parameter fields. A parameter field is always denoted in angle brackets ('<' and '>'). A parameter field is often a list of options from which the user must choose when instantiating a requirement. In some cases, it is not a list of options but a field in which a user can enter free text. For example, if a field expects some number, there are infinitely many options. These cannot all be listed, but FRET will check whether a number has actually been entered in order to formalize it.

In turn, any of the selected options can be new parameter fields, so we refer to `Trigger`, `InitialConditions`, `PostConditions`, and `Time` as the `original` parameter fields of the CEL template. These original parameter fields consist of a field type, an representation (optional), and a name:

$$< \texttt{TYPE}[\texttt{REPRESENTATION}] : \texttt{NAME} >$$

Type denotes the `Field type`, which is explained in the next item. The representation is what is shown to a user. If no representation is given, the name will be shown. The name is what is used to refer to the field in the rest of the template. For example, for `InitialConditions` the field specification is $< \texttt{Co}[\texttt{InitialConditions}] : \texttt{IC} >$.

This means that `InitialConditions` is a combo box (`Co`), which is a type of dialogue box that lists a number of values. Furthermore, the prompt that the user sees for the combo box is ‘InitialConditions’, because this was entered as the representation of the field. Finally, within the rest of the template and the requirements that use this template, this field is referred to as `IC`. For example, the `Field types` that are discussed next relate a field’s name to its possible types.

Field types The field types map the parameter fields to a list of possible choices for those fields. For example, they map `InitialConditions` to three options:

- < `Any` >
  FRET interprets this as a text field in which any input can be added.
- *true*
  FRET interprets this as a static field that represents the logical condition `true`.
- < `Parser` : `Condition` >
  FRET interprets any field of the form < `Parser` : `PARSERNAME` > as a text field combined with a parser, in which the user is expected to type text that matches the given parser. In this case, the parser `Condition` is used, which expects text in the format of a logical condition. A colored circle is shown next to the text field; it is green if a correct logical condition is entered and red otherwise. A requirement can only be formalized if all of its parameter fields match any of the possible parser or other fields.

There are many more field types besides the ones mentioned above, the details of which will not be discussed here. If the user selects one of the options, FRET will interpret it and show it instead of the original parameter field.

Questions The optional additional phrases are transformed into questions, with a list of answers from which a user can choose. For example, the `Persistence phrase` is transformed into:

> Do the Initialconditions need to be true only at the time where the Trigger is true?

When a user instantiates one of the parameter fields, this might influence the questions. For example, if the `Time` is 1, the `PostConditions` are true immediately. So, in this case, there is no reason to still answer this question. Therefore, FRET will disable this question and no longer show it to the end user. The mapping between field answers and questions is stored in `Default field answers`, explained next.

Default field answers A mapping between certain field answers and questions. In the example explained previously, there would be a mapping between `Time` being 1 and disabling the Persistence question. However, each question must be answered, because each distinct set of answers is mapped to a formalization. In this case, the `InitialConditions` being true only at the time where the `Trigger` is true is the same as the `InitialConditions` persisting until the `PostConditions`. Therefore, it is irrelevant which answer is chosen, so we choose to automatically select 'Yes'. This boils down to the following:

$$\textbf{Time} = 1 \implies \textbf{Q2} = \text{Yes}$$

4.1.2 Creating a requirement

When a user wants to create requirements for a system, the first step is to select a workspace and create a project, because all configurations are project-wide. The newly created project will open an empty list of requirements, to which new requirements can be added. To create a requirement, the user must select the template that captures it most accurately. As an example, we will use the following requirement from challenge 1:

> "Exceeding sensor limits shall (immediately) latch an autopilot pullup when the pilot is not in control and the system is supported without failures."

The Simulink model of challenge 1 uses the following boolean input and output signals:

limits sensor limits are exceeded (input)

standby the pilot is in control (input)

apfail there are failures (input)

pullup an autopilot pullup is latched (output)

For this requirement, we will select the CEL template. It requires the user to enter values for the fields `TR`, `IC`, `PC`, and `TI` as given in Table 4.2 and to answer the two questions. The requirement can be instantiated as follows:

TR – Select option `true`

IC – Select option <`Parser:Condition`>

Id	1		
Template	Conditional Event Logic		
Fields	TI	true	
	IC	limits==true and standby==false and apfail==false	
	PC	pullup==true	
	TR	1 frames	
Answers	If a new Trigger occurs, does it release the obligations set by the previous Trigger?		No
	Do the InitialConditions need to be true only at the time where the Trigger is true (as opposed to persist until the PostConditions)?		Yes

TABLE 4.3: FRET requirement based on Conditional Event Logic template

- Enter `limits==true and standby==false and apfail==false` in the text field that appears in the place of the combo box

PC
- Select option <`Parser:Condition`>
- Enter `pullup==true` in the text field that appears in the place of the combo box

TI
- Select option `1 frames`

Since there is no actual `Trigger` specified and the `Time` is 1 (thus immediately), both questions are answered automatically and thus hidden.

This requirement can be saved in the format given in Table 4.3.

After creating a requirement, it can be edited or removed at all times.

4.1.2.1 Free Form template

FRET is targeted towards requirement elicitation and automated formalization through the template basis. However, there might be situations in which users do not want to base their requirements on a template and formalize them, yet they do want to use a single tool for all of their requirements. Therefore, FRET also offers a template called *Free Form*. This template is a single text box in which users can type anything. It cannot be formalized due to lack of structure and ambiguity. This allows users to store all their requirements in FRET, even if they do not wish to use templates. In combination with the hierarchy feature that is discussed in the next section, the Free Form template can also be used to define a high-level requirement, while providing more detail in the children of that requirement.

4.1.2.2 Hierarchy

FRET offers a capability for hierarchy within the set of requirements for a system. Each requirement can have multiple children and at most a single parent. The intended purpose of this hierarchy is that the children are a more detailed specification of the parent. However, FRET does not interpret this hierarchy in any way: it is purely a way for users to provide a better overview. Hierarchy can be used in a number of ways:

- **Increase level of detail**
  A fully unambiguous requirement might be too specific to be readily understood. Therefore, hierarchy can be used to provide different levels of detail for a requirement. The top-requirement might be a high-level requirement that is easy to understand, with its child being more detailed and less ambiguous. This child might have a child in turn that provides even more detail, until the lowest level is fully unambiguous. For example, the following requirement can be specified in free form:

  > In the no-fail state, the mid-value shall be the selected value.

  It can have the following child in the CEL template:

  > Whenever true and previous(FC) == 0, then sel_val == max(min(a,b), min(max(a,b),c)) must become true within 1 frames.

  These two have the same meaning, but clearly the free form representation is much easier to read. On the other hand, the CEL representation is much easier to formalize.

- **Decompose into multiple sub-requirements**
  A requirement can be composed of multiple sub-requirements. Using hierarchy, both the high-level requirement and its sub-requirements can be specified in FRET. For example, the following requirement can be specified in free form:

  > In the no-fail state, a miscompare, which shall be characterized by one branch differing with the other two branches by a unique trip level that lasts for more than the persistence limit, shall be reported to failure management as a failure.

  It can have the following children in the CEL template:

  > Whenever true and previous(FC)==0 and |ia-ib| >Tlevel and |ia-ic| >Tlevel and PC>PClimit, then FC==4 must become true within 1 frames.

> Whenever true and previous(FC)==0 and |ib-ia| >Tlevel and |ib-ic| >Tlevel and PC>PClimit, then FC==2 must become true within 1 frames.
>
> Whenever true and previous(FC)==0 and |ic-ia| >Tlevel and |ic-ib| >Tlevel and PC>PClimit, then FC==1 must become true within 1 frames.

This would have become a very long and complex requirement if we had not split it into three components. Without hierarchy, the connection between these subrequirements would have been lost.

The above are examples of how hierarchy can be used in FRET. However, FRET does not assign any semantics to this functionality, so users are free to use their own interpretations to make their requirements specifications more organized. Basically, FRET is capable of formalizing any requirement that contains only parameters fields that are parsable by FRET (for example, no requirements that are based on the free form template). A user can select specific requirements for formalization; this will not formalize their parents or children. However, logically, only leaf children should be formalizable, because they are normally more detailed versions of their parents, which means that its parent is not fully unambiguous.

4.1.3 Domain patterns

A template such as 'Conditional Event Logic' captures many requirements. However, it might not be the most intuitive template to use in all cases. For some projects, several requirements might be similar, so that some field always has the same value. For this purpose, FRET offers the possibility to specify *domain patterns*. A domain pattern is a rephrased and/or more detailed version of a template that resembles the requirements in its domain more closely than its target template does.

A domain pattern is just another template that (partially) instantiates an existing template. It has its own `Name`, `Fields`, and `Field types`, but inherits the `Questions` and `Default answers` from its target template. Templates also have a number of additional fields, that are only specified for domain patterns:

Target template The template that the domain pattern extends, such as 'Conditional Event Logic'.

Mapping A mapping of the fields of the domain pattern template to its target template.

An example of a domain pattern is the *Select Value* template, given in Table 4.4. This domain pattern targets a specific system that has the following outputs:

Name	Select Value	
Fields	If the failure state is	
	<**Co[FaultCode]:S**>	
	and	
	<**Co[InitialConditions]:C**>	
	, then the selected value will be set to	
	<**Co[SelectedValue]:V**>	
	.	
Field types	S	0
		1
		2
		4
	C	<Any>
		true
		<Parser:Condition>
	V	<Parser:Arithmetic>
Target template	Conditional Event Logic	
Mapping	TR	true
	IC	previous(FC)==[S] and [C]
	PC	sel_val==[V]
	TI	1 frames

Table 4.4: FRET template for Select Value domain pattern

- `FC`

  The fault code of the system, either 0, 1, 2, or 4.

- `sel_val`

  The value that the system selects, depending on the current fault code.

In terms of the CEL template, this domain pattern states the following:

> Whenever [true] and [previous(FC)==FaultCode], then
>
> [sel_val ==SelectedValue] must become true within [1 frames].

If the Select Value template is used, the user only needs to select a `FaultCode` and enter an arithmetic expression for the `SelectedValue`. On the other hand, if the CEL template is used, the user must enter all the expressions given within brackets.

4.2 Requirements Formalization

The concept of requirement templates can be used for automated formalization of requirements. If a template is unambiguous, it can be formalized. Therefore, besides the fields mentioned before, FRET templates contain an additional field.

Formalizations A list that contains a *formalization* for each format that is specified for this requirement template. Each formalization consists of the following components:

Name The name of the formal language that this formalization will formalize to, for example 'LTL' or 'CoCoSim'.

Question mapping A unique combination of answers to the template questions. Since the purpose of the questions is to make a template unambiguous, different sets of answers map to different formalizations. For example, if a users selects Q1='Yes' and Q2='Yes', FRET will use a different formalization template than if the user would have selected Q1='Yes' and Q2='No'. This will be demonstrated in the next section.

Formalization template The representation of the requirement template in the formal language that this formalization corresponds to. This formalization template contains the same to-be-instantiated fields as the requirement template. When the requirement template is selected by a specific requirement, these fields will be instantiated. Basically, the formalization template formalizes the requirement template. However, the actual fields are still not formalized at this point. This is where the parsers come in, which is discussed in section 4.2.2. If the contents of an instantiated fields do not match any of the parsers that were specified for that field, the requirement cannot be formalized. FRET does not store which field type was selected by a user, only if it matches any of the possible parsers. For example, in Section 4.1, we discussed that `InitialConditions` could have three possible field types: $<$ `Any` $>$, `true`, and $<$ `Parser` : `Condition` $>$. So, the only possible parser is `Condition`. If a user selects $<$ `Any` $>$ and types '`This is any text.`', FRET will try to parse this with the `Condition` parser and fail. However, if a user select $<$ `Any` $>$ and types '`a>1`', the `Condition` parser will succeed and the requirement can be formalized. Clearly, it would have been more useful for the user to immediately select $<$ `Parser` : `Condition` $>$ as the field type, because FRET will then give instant feedback on whether or not the input is actually a `Condition`, but FRET does not restrict the user in this way. The same goes for the case when the user selects `true`. Upon formalization, FRET will simply parse this as a `Condition`.

This `Formalizations` field is only visible in the source template file and is not shown in the GUI to the user.

If a template is formalized, this means that it is transformed from a natural language FRET template into a selected formal language, such as a CoCoSim requirement or

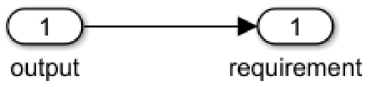


FIGURE 4.2: Simple CoCoSim requirement

```
add_block(simulink/Commonly Used Blocks/In1, 'Requirement/output');
add_block(simulink/Commonly Used Blocks/Out1, 'Requirement/requirement');
add_line('Requirement', 'Requirement/output/1', 'Requirement/requirement/1);
```

LISTING 4.1: Matlab code to generate CoCoSim requirement

MTL formula. Each FRET template contains distinct formalizations (the formalization templates) for each combination of answers to the template questions (question mapping), because these questions were added specifically to specify the exact semantics of a template. Thus, they have a big impact on the resulting formalization.

The template itself can be formalized by retrieving the answers to the template questions that a user chose for a requirement and returning the corresponding formalization template. This formalization template still contains the empty parameters for the parameter fields that have been instantiated in the requirement. These are separately transformed using parsers and replace the empty parameters in the formalized template.

4.2.1 Formalization to CoCoSim

This section discusses formalization of templates to the CoCoSim format. This formalization has been specified for the CEL template. Because the CEL template contains two questions, there are four distinct formalization templates possible:

- Q1='No' and Q2='Yes' (NY)
- Q1='Yes' and Q2='Yes' (YY)
- Q1='Yes' and Q2='No' (YN)
- Q1='No' and Q2='No' (NN)

For each formalization template, FRET stores a textual format in the requirement template. For CoCoSim, this is the Matlab code that must be executed in order to create the corresponding Simulink model. This is thus basically a textual representation of the model. For example, if we want to create a requirement in CoCoSim called `Requirement` and we want to specify that the requirement directly depends on an output called `output`, we need the Matlab code given in Listing 4.1 to generate it.

Basically, this requirement states that the model is only correct if the output is always true. Line 1 creates the input block called ‘output’ in subsystem ‘Requirement’ (Note that outputs of the model are actually considered inputs of the requirement subsystem). Line 2 creates the output block called ‘requirement’ in subsystem ‘Requirement’. Finally, line 3 adds a line between the first (and only) out port of ’input’ and the first (and only) in port of ‘requirement’. Figure 4.2 shows the result.

In this example, we do not show the creation of the requirement subsystem block or the additional block parameters that need to be set, to keep it as simple as possible. However, for realistic requirements, this code can become quite long and complex and does not provide a good overview. For this reason we illustrate formalization to CoCoSim using the generated Simulink models instead of this intermediate textual format, throughout the rest of this chapter.

A general CoCoSim formalization of the CEL template (Table 4.2) is given in Figure 4.3. Since the model is autogenerated, it has been reformatted to provide a better overview. The red block `deactivate` depends on the answers to the questions, which will be discussed later.

The validity of any requirement based on this template is determined by maintaining a timer while a requirement is *active*. A requirement is *activated* if the `Trigger` and `InitialConditions` are true at the same time. The functionality is split up into three steps:

1. *Is the requirement active?*
   The requirement is *active* if the `Trigger` and `InitialConditions` are true at the same time (`TR and IC`), or if it was already active and has not been *deactivated* (`prev_time` $>$ `0 and not deactivate`). It is deactivated if something happens to release its obligations, during the time that the requirement is active. What this could be specifically depends on the answers to the questions. Additionally, a requirement is only active while `PC` is false, because otherwise, it has been satisfied (`not PC`).

2. *Update active time.*
   If the requirement is deactivated, we set the timer to 0. Otherwise, we set the timer to be the previous time. Then, if it is currently active, the timer is incremented. If it is not active, the timer is reset.

3. *Is the requirement satisfied?*
   Because the timer keeps track of how long a requirement has been active, and thus the time during which the `PostConditions` have not been satisfied, since `Trigger`

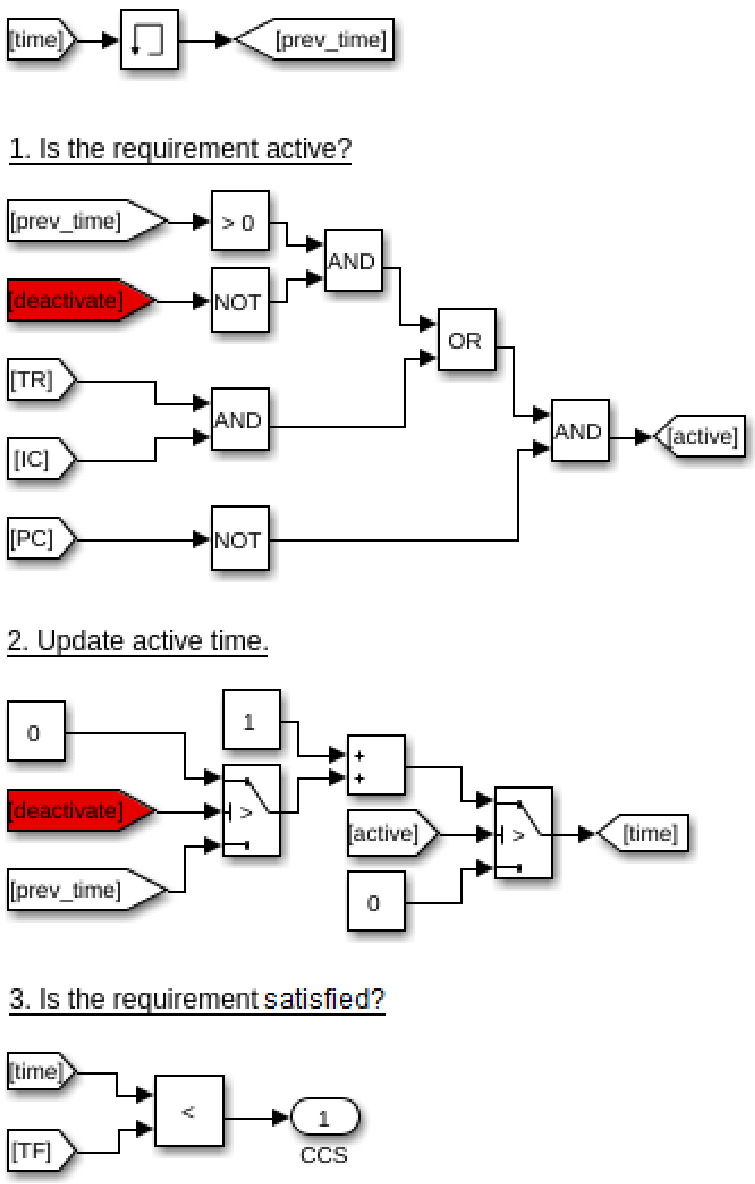
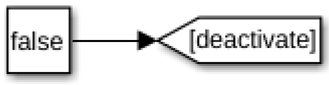


FIGURE 4.3: CoCoSim formalization of the CEL template

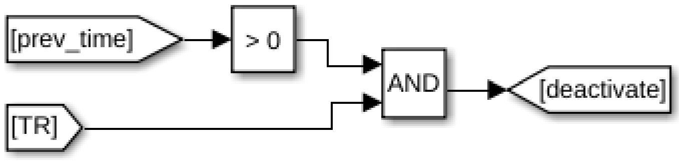
and `InitialConditions` were true, we check whether the value of the timer is still less than `Time`. If this is always the case, the requirement is satisfied. Otherwise, a counterexample exists.

As discussed, four different formalization templates exist for the CEL template, that depend on the answers to the two questions. The contents of the `deactivate` block depend on this.

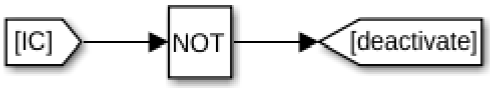
For formalization NY, the `deactivate` block is always `false`. As soon as `Trigger` and `InitialConditions` are true, `PostConditions` must be true within `TI`. There is no way to release this obligation, thus the requirement cannot be deactivated.



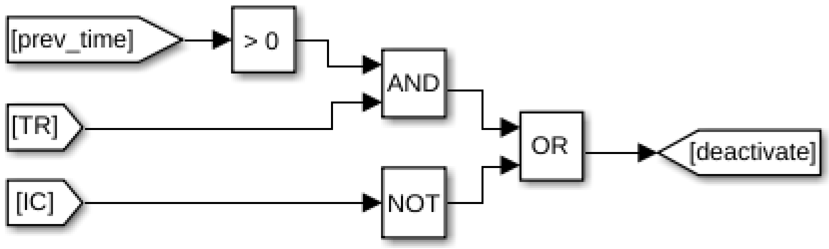
(A) Formalization NY



(B) Formalization YY



(C) Formalization NN



(D) Formalization YN

FIGURE 4.4: Contents of the `deactivate` block

For formalization YY, the `deactivate` block is equal to `prevTime > 0 and TR`. This means that if another `Trigger` occurs, it will release the obligations and thus deactivate the requirement (thus a `Trigger` that occurs after the requirement is already active, to prevent a `Trigger` from releasing its own obligations).

For formalization NN, the `deactivate` block is equal to `not IC`. This means that if the `InitialConditions` stop being true while a requirement is active, this releases the obligations and thus deactivates the requirement.

For formalization YN, the `deactivate` block combines the YY and NN formalization, so it is equal to: `(prevTime > 0 and TR) or IC`.

For each of these formalizations, the `deactivate` block is illustrated in Figure 4.4.

4.2.2 Parsers

The formalization templates map a formalization to a requirement template. However, this approach does not take into account the fields that are instantiated by a requirement. This is where the `Parser` field type comes into play. A parser can be used to make sure that a user only enters values into fields that can be understood in the context of the

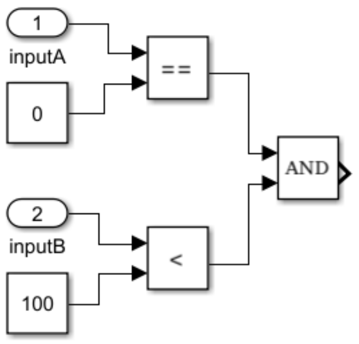


FIGURE 4.5: CoCoSim formalization of `InitialConditions`

requirement. For example, in order to formalize a requirement based on the CEL template, a user should enter some sort of condition into field `C` for the `InitialConditions`, such as:

$$\texttt{inputA} == 0 \text{ and } \texttt{inputB} < 100$$

At this point, the parser only checks the input, but does not build any sort of data structure from the input. However, when a requirement is formalized, the contents of the fields must be transformed into the same format as the formalization template.

For this purpose, the parsers can be extended for each formalization type in order to transform the fields into the expected format. We call these extensions *transformers*. For example, the CoCoSim formalization will transform the `InitialConditions` given above (through the intermediate format) into the Simulink submodel given in Figure 4.5.

4.2.3 Formalizing a requirement

To formalize a selected requirement, the first step is to choose a formalization type. Then, a textual format of the formalized requirement will appear immediately. Depending on the type of formalization selected, additional options might be available to the user.

For example, if formalization to CoCoSim is selected and if Matlab is available, a Simulink model containing the requirement can be created directly from within FRET. In order to maintain independence between the requirement and the Simulink model it belongs (the *base model*), a separate Simulink model is created for the requirement. The user can also enter the path to the subsystem of the base model in which the requirement is supposed to be inserted. At any later point, the requirement model and the provided subsystem path can be used to insert the requirement into the existing model and attempt to verify the model with regard to that requirement using CoCoSim.

4.3 Configurability

So far, what has been discussed about FRET is only useful for requirements that correspond to a predefined template, using only predefined parsers, and that need only be formalized into any of the predefined formalizations. However, a major feature of FRET is its configurability. Because of this, FRET is potentially suitable for specification and formalization of any type of requirement and project. This is what sets it apart from any existing work.

FRET is designed to be as configurable as possible. This includes simple changes, such as to the user interface, but also changes to its core functionality. FRET allows subject matter experts to adapt the environment to their specific subject domain. This means that custom templates, parsers and formalizations can be loaded into FRET.

4.3.1 Templates

Every project contains a configuration folder that contains a configuration file with all templates that are available for that project. A subject matter expert can edit the existing templates in this file or add new templates to it. This simply requires specifying all the template fields that are listed in Section 4.1.1. The updated list of templates is immediately shown upon requirement creation. Currently, this requires that subject matter experts edit and add templates in JSON.

A subject matter expert can add an entirely new template to capture requirements that could not otherwise be captured, or create a domain pattern that targets an existing template. The functionality of domain patterns is an extension of the Core phrases that Dwyer et al. [26] offer. In FRET, different domain patterns that target the same template are not only different formulations of the same phrase, but can also have varying degrees of instantiation, depending on how much information is fixed in the domain. The most important difference is that these domain patterns are not predefined by FRET, but teams can create their own templates and extensions of templates. Thus, they are not limited to the templates and phrasing that were devised by the creators of the requirements specification tool.

4.3.2 Parsers

Similarly, every project contains a folder into which custom parsers can be placed. Currently, this does require users to create parsers in the Scala FastParse library [42], in which the default parsers are created, so specific knowledge is required. These user-supplied parsers are loaded by FRET and template fields can use them as additional field types.

4.3.3 Formalizations

Every template that does not target another template has its own list of formalizations. For example, the CEL template contains formalizations to LTL and CoCoSim by default. If another type of formalization is required, the corresponding formalization template can simply be added to the existing requirement template. This might also require that the parsers that are used by the template are extended for use by this specific formalization (see section 4.2.2), including corresponding transformers to chosen formalizations. However, this specific functionality has not been implemented yet. To add a formalization to a template, a subject matter expert must be familiar with the format in which that formalization type must be specified.

Part III

Evaluation

Chapter 5

Performance

FRET should be suitable for the specification and formalization of a wide range of requirements. We consider the performance of FRET to be the ratio of supported requirements versus the number of requirements that FRET cannot support (yet), but that are manually formalizable. This is evaluated through the second case study for which the LM challenges are used again, which is compared to the first case study, that was used to evaluate the current situation of requirements specification and formalization. This will indicate how well FRET performs compared to this current situation.

5.1 Specification

Table 5.1 shows in green which of the requirements could be successfully specified in FRET. Clearly, all requirements could be specified in FRET. This is partly due to the 'Free Form' requirement. Even if we cannot define a formal template that captures a certain requirement, it can still be specified in some way in free form format. This will not allow the requirement to be formalized. However, it makes it possible to use FRET for the specification of all requirements, even if the purpose is not to formalize them or if they are supposed to be specified further at a later stage.

5.2 Formalization

Table 5.2 shows which requirements could actually be formalized into CoCoSim requirements through FRET.

		Requirements												
		1	2	3	4	5	6	7	8	9	10	11	12	13
Challenges	0	🟩	🟩	🟩	🟩									
	1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	2	🟩	🟩	🟩	🟩									
	3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩			
	4	🟩	🟩	🟩	🟩	🟩	🟩	🟩						
	5	🟩	🟩	🟩	🟩									
	6	🟩	🟩	🟩	🟩	🟩								
	7	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩		
	8	🟩	🟩											
	9	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩					

TABLE 5.1: Specification of requirements in FRET

		Requirements												
		1	2	3	4	5	6	7	8	9	10	11	12	13
Challenges	0	🟩	🟩	🟩	🟩									
	1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
	2	🟩	🟩	🟩	🟩									
	3	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩			
	4	🟧	🟧	🟧	🟧	🟧	🟧	🟧						
	5	🟩	🟩	🟥	🟥									
	6	🟧	🟧	🟧	🟧	🟧								
	7	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧		
	8	🟩	🟩											
	9	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧					

TABLE 5.2: Formalization of requirements through FRET

The green cells indicate the requirements that could be formalized by FRET. Some of these requirements were ambiguous and needed to be analyzed and changed before manual formalization was possible (the orange cells in Table 3.2), as discussed in Chapter 3. These issues were resolved through the unambiguous specification of requirements that FRET offers. For example, the following requirements of challenge 1 were problematic:

2. The autopilot shall change states from TRANSITION to STANDBY when the pilot is in control.

3. The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.

By analyzing these requirements and the system description, we figured out that requirement 3 needed the additional initial condition that 'the pilot is not in control'.

In FRET, we prevented this ambiguity by defining a domain pattern that requires the user to enter the required value of every input. We defined the following template:

> The autopilot shall change states from <PrevState> to <NextState> whenever standby is <Standby> and limits is <Limits> and supported is <Supported> and apfail is <Apfail> and good is <Good>.

The parameters provide the following options:

- PrevState: 0 | 1 | 2 | 3
- NextState: 0 | 1 | 2 | 3
- Standby: true | false | standby
- Limits = true | false | limits
- Supported = true | false | supported
- Apfail = true | false | apfail
- Good = true | false | good

Now, the user is forced to consider the value of each input that is required as an initial condition for the requirements. For example, when the user specifies requirement 3 of challenge 1, the instantiation 'PrevState=0, NextState=3, Standby=true, Limits=limits, Supported=supported, Apfail=apfail, and Good=good' must be entered. FRET does not assign any semantics to 'limits', 'supported', 'apfail', and 'good' as they are used here. However, this template targets the CEL template, so it is mapped to this template as:

> Whenever true and state==0 and standby==true && limits==limits && supported==supported && apfail==apfail && good==good, then STATE==3 must become true within 1 frames.

So, for example for 'limits==limits', both versions of limits are the same as far as FRET is concerned, so this will be equivalent to 'true' if it is formalized.

The orange cells are the requirements that could not be formalized manually, due to a missing library or lack of background knowledge (the red cells in Table 3.2). Because, of this, we were not able to unambiguously specify these requirements in FRET. Instead, we used the Free Form template to be able to specify them in FRET nonetheless, without being able to formalize them. If the requirements of challenges 4 and 7 (for which we did not have enough background knowledge) had been specified in FRET from the start, this problem could have been prevented. This would have forced the domain experts

Challenge	0	1	2	3	4	5	6	7	8	9
# Domain patterns	2	3	0	2	-	1	-	-	0	-

Table 5.3: Number of domain patterns created per challenge

to specify the requirements unambiguously, thus not requiring domain expertise during formalization.

The cells in red are the requirements that could be formalized manually, but not (yet) by FRET. These are the following requirements of challenge 5:

1. Using a first order finite backward difference equation, the spatial derivatives of $\Delta z/\Delta xt = (z(n,1) - z(n-1))/(xt(n,1) - xt(n-1,1)$ and $\Delta z/\Delta yt = (z(n,1) - z(n-1))/(yt(n,1) - yt(n-1,1)$ shall never exceed a top bound of +10 or bottom bound of -35 (e.g. $-35 <= \Delta z/\Delta(xt, yt) <= 10$), where n denotes an index to the current values and $n-1$ denotes the prior values in the included truth data for xt and yt.
2. The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, for the equivalent input of (xt, yt).

These requirements both use time series constants, which are not currently supported by FRET. For example, z is a time series constant that supplies its next value at each time step. FRET can only read single-value constants as of right now.

All of the requirements that can be formalized by FRET are mapped to the CEL template. However, domain patterns were created for each project as intermediate templates to target the specific domain and to demonstrate FRET's customization capabilities. These domain patterns are purely for demonstration purposes: each subject matter expert is free to design their own domain patterns. Table 5.3 gives the number of domain patterns that we created for each challenge. The actual domain pattern templates are given in Appendix B. The instantiated requirements are given in Appendix C.

In total, 34 out of 68 requirements could be formalized by FRET. However, only 36 could be formalized by us manually, so FRET is currently capable of formalizing $34/36 = 94\%$ of the 'formalizable' requirements.

Chapter 6

Usability

To determine the usability of FRET, we compare our experiences of the two case studies. The first case study provides an evaluation of manual requirement formalization, discussed in Section 3. The second case study evaluates the use of FRET for requirement formalization, discussed in Section 5.

Our findings are given in the following sections.

6.1 Time and effort

There is little time and effort required for simply writing down the requirements in a natural language. However, making sure that they are unambiguous and contain sufficient detail is often not done properly. Then, if someone wants to formalize these natural language requirements manually, it can take up a lot of time to determine what exactly they mean. In the case of the LM challenges, it actually took weeks to first understand what the requirements meant in terms of the corresponding Simulink model and then even more weeks to remove any ambiguity and make sure that the requirements did not contradict each other.

For formalization, the user also needs to specify each requirement again, but this time in the selected formalism. In the case of CoCoSim, it takes up much more time to create and connect all the blocks than it does to write down a requirement in a textual format. However, until FRET, this was still the least tedious way to create CoCoSim requirements.

On the other hand, when using FRET, most of the time and effort put into the specification of requirements is regarding the creation of new templates and parsers. However, this is done infrequently on a per-project basis by subject matter experts. Furthermore, mostly domain patterns will be created, which do not require much time and effort at all. The actual specification of requirements does not require much time and effort, since the end user only needs to select a template and fill in the fields. It might take a number of weeks to train subject matter experts and hours or days to come up with and specify suitable templates, but the final specification of a requirement in FRET by end users can be done in a matter of minutes. Of course, this only takes into account the time for writing down the requirements, so it assumes that the users already have in mind what the requirements should specify.

The time and effort required for formalization through FRET into an existing formalism is negligible, because the end user only needs to select the formalization and the rest is done by FRET. On the other hand, the time and effort required for creating a new formalization is more significant. However, this only needs to be done once for every template that is not a domain pattern, as opposed to once for every separate requirement when manual formalization is used.

6.2 Error-proneness

In Section 3.3, we discussed an issue with the specification of requirements 2 and 3 of challenge 1, that could lead to serious misunderstandings. The problem was that requirement 3 assumed information that was given in requirement 2, but did not explicitly state this. If these requirements are formalized without first analyzing their exact meaning, requirement 3 would result in a counterexample.

This shows how error-prone the natural language specification of requirements is. What seems obvious to the person specifying a requirement, might be overlooked by the person responsible for formalizing this requirement, resulting in a requirements that does not match its intended meaning. In FRET, situations such as these can be prevented. However, the error-proneness of FRET highly depends on the templates. It would be entirely possible to specify a requirement, while leaving out that a certain input must have a certain value, because it seems obvious in the context. FRET will be capable of formalizing this requirement, because it is syntactically correct (and it will in fact be unambiguous), but it will not capture the correct meaning.

However, it is the responsibility of the subject matter expert to minimize the risk of such mistakes. In this case, for example, the subject matter expert can create a template that requires the end users to specify the value of each and every input when instantiating a requirement. This was actually done in the second case study. The resulting templates and requirements can be found in Appendices B and C.

6.3 Ease of use

It is very easy to simply specify requirements in natural language. However, it is a lot harder to do this correctly and unambiguously. The ease of formalization entirely depends on the formalism that is used. This can be very easy or involve a steep learning curve.

FRET is targeted towards end users, so the result is that it is very easy to use when specifying or formalizing requirements. The complexity is to be dealt with by the subject matter expert. The subject matter expert is responsible for defining (domain pattern) templates, parsers, ontologies, and formalizations. This requires a familiarity with FRET and its underlying technologies. However, the goal is to make this easier, by separating configurable aspects of FRET from its underlying implementation. Chapter 9 discusses this further.

Chapter 7

Consistency

The final research topic is about the correctness of the formalizations performed by FRET. Ideally, we would interpret the meaning of a template directly and compare it to its formalizations. However, this would require advanced natural language processing, which is not the focus of this research. Instead, we focused on the consistency of template formalizations.

Because consistency is a broad term that could also refer to the mutual consistency between requirements (not contradicting each other), we will give a definition of the meaning of consistency as an evaluation metric for FRET:

> *A template in FRET is consistent if the transformation of a single template into different formalization types result in equivalent formalizations.*

As discussed before, we have defined a CoCoSim formalization for the CEL template. Additionally, an MTL formalization exists for the CEL template. We have compared these formalizations to each other. Are they equivalent? Only in that case, we can establish that the CEL template is consistent.

An MTL formula and a CoCoSim requirement cannot be compared directly, so we had to use an intermediate format. CoCoSim automatically generates Lustre code from a given CoCoSim requirement, so we used Lustre as the intermediate format.

This intermediate Lustre file consisted of three nodes:

1. A Lustre representation of the requirement in CoCoSim (the CoCoSim node).
2. A Lustre representation of the requirement in MTL (the MTL node).
3. An equivalence check of the previous two nodes (the equivalence node).

The first node is provided by CoCoSim. The translation from MTL to a Lustre node was done manually, as was the creation of the third node.

As a reminder, the Core phrase of the CEL template is:

> "Whenever `TR` and `IC`, then `PC` must become true within `TI`."

This can be formalized into MTL as follows:

$$\mathbf{G}\ ((\texttt{TR}\ \&\&\ \texttt{IC}) \Rightarrow \mathbf{F}[1,\texttt{TI}]\ \texttt{PC})$$

In other words: if at any time `TR` and `IC` are both true, eventually, within `TI` time steps from that time, `PC` must become true. To transform this into Lustre, we must keep track of the time that has passed since `TR` and `IC` were true simultaneously, until `PC` is true. If this time ever exceeds `TI`, the requirement is not satisfied. Otherwise, it is. We say that the requirement is *active* if `TR` and `IC` are true or have been true in the past, and `PC` has not been true since. Thus, the timer gets incremented during active periods.

When transforming this to Lustre, if we consider any random time step, one of three cases must be true:

1. *The requirement is not active.*
   Either `PC` is true (the requirement is deactivated) or he requirement was not active yet and `TR` and `IC` are not currently true (the requirement has not been activated). The timer is set to 0.

2. *The requirement becomes activated during this time step.*
   The requirement was not active yet, but currently `TR` and `IC` are true and `PC` is not true. The timer is set to 1.

3. *The requirement has been active for a while.*
   The requirement was already active and `PC` is not true. The timer is incremented.

This requires that we maintain two variables besides the timer: whether the requirement continues to be activates, if it was active before, and, if not, whether it is activated during this time step. The corresponding Lustre node is given in Listing 7.1.

However, the MTL formula is further specified by the questions:

Q1 If a new Trigger occurs, does it release the obligations set by the previous Trigger?

Q2 Do the InitialConditions need to be true only at the time where the Trigger is true (as opposed to persist until the PostConditions)?

```
node MTL(TR:bool; IC:bool; PC:bool; TI:int)
returns(OK:bool);
var
    activate:bool;
    continue:bool;
    timer: int;
let
    activate = TR and IC and not PC;
    continue = false -> (pre activate or pre continue) and not PC;
    timer = if continue then pre timer + 1 else (if activate then 1 else 0);
    OK = timer < TI;
tel
```

Listing 7.1: Lustre node for CEL Core phrase

As explained in Section 4.2.1, the CEL template is split up into four different formalization templates (NY, YY, YN, and NN) for each formalization type. The CoCoSim templates was discussed previously. In the next section, we discuss the corresponding MTL templates. Next, we discuss how we established the equivalence of any pair of CoCoSim and MTL templates, after which we provide the results.

7.1 MTL nodes

7.1.1 NY

The NY template does not release the obligations of a previous `Trigger` if a new `Trigger` occurs and does not require that the `InitialConditions` persist until the `PostConditions`. This means that if at any time `TR` and `IC` are both true, eventually, within `TI` time steps from that time, `PC` must become true, no matter what happens with `TR` and `IC` after activation. This exactly matches the CEL Core phrase, so the MTL formula is the same:

$$\mathbf{G}\ ((\texttt{TR}\ \&\&\ \texttt{IC}) \Rightarrow \mathbf{F}[1,\texttt{TI}]\ \texttt{PC})$$

So, the Lustre node for the MTL NY formalization template is already given in Listing 7.1.

7.1.2 YY

The YY template releases the obligations of a previous `Trigger` if a new `Trigger` occurs and does not require that the `InitialConditions` persist until the `PostConditions`.

This means that if at any time `TR` and `IC` are both true, eventually, within `TI` time steps from that time, `PC` must become true. However, if `TR` occurs again before this time frame is over (so between 2 and `TI` time steps), these obligations are released.

This means that the MTL YY formalization template is slightly different than the NY formalization template:

$$\mathbf{G}\ ((\texttt{TR}\ \&\&\ \texttt{IC}) => (\mathbf{F}[1,\texttt{TI}]\ \texttt{PC}\ ||\ \mathbf{F}[2,\texttt{TI}]\ \texttt{TR}))$$

The corresponding Lustre node is the same as the one for the NY template, except that the timer should no longer continue if `TR` becomes true while the requirement is active. So, we replace the assignment of `continue` by:

```
continue = false −> (pre activate or pre continue) and not PC
    and not (TR and pre timer > 0);
```

The underlined part denotes where this line is different from Listing 7.1. This addition checks whether no `Trigger` occurs, or if it does, whether this is actually a new `Trigger` (if `pre timer` is not greater than 0, the occurring `Trigger` is actually the one that activates the requirement and it should not release its own obligations). If a new `Trigger` has actually occurred, `continue` is set to false and the requirement is deactivated. It may or may not be activated again immediately, depending on the value of `IC`.

7.1.3 NN

The NN template does not release the obligations of a previous `Trigger` if a new `Trigger` occurs and requires that the `InitialConditions` persist until the `PostConditions`. This means that if at any time `TR` and `IC` are both true, eventually, within `TI` time steps from that time, `PC` must become true. However, if `IC` becomes false before `PC` has become true, these obligations are released.

This means that the MTL NN formalization template is also slightly different than the NY formalization template:

$$\mathbf{G}\ ((\texttt{TR}\ \&\&\ \texttt{IC}) => (\mathbf{F}[1,\texttt{TI}]\ (\texttt{PC}\ ||\ !\texttt{IC})))$$

The corresponding Lustre node is the same as the one for the NY template, except that the timer should no longer continue if `IC` becomes false while the requirement is active. So, we replace the assignment of `continue` by:

```
continue = false −> (pre activate or pre continue) and not PC and IC;
```

The underlined part denotes where this line is different from Listing 7.1. This addition checks whether `IC` does not become false while the requirement is active. If it does, `continue` is set to false and the requirement is deactivated.

7.1.4 YN

The YN template releases the obligations of a previous `Trigger` if a new `Trigger` occurs and requires that the `InitialConditions` persist until the `PostConditions`. This combines the two alterations of the `NY` and `NN` formalization templates.

So, the MTL YN formalization template also combines the changes of the YN and NN templates to the NY formalization template:

$$\mathbf{G}\ ((\texttt{TR}\ \&\&\ \texttt{IC}) => (\mathbf{F}[1,\texttt{TI}]\ (\texttt{PC}\ ||\ !\texttt{IC})\ ||\ \mathbf{F}[2,\texttt{TI}]\ \texttt{TR}))$$

So, the corresponding Lustre node changes the assignment of `continue` to:

```
continue = false -> (pre activate or pre continue) and not PC
   and not (TR and pre timer > 0) and C;
```

7.2 Equivalence node

The equivalence node compares whether any CoCoSim node and a supposedly equivalent MTL node output the same values for all the same inputs. The Lustre code for the equivalence node is given in Listing 7.2

```
node equivalence(TR : bool; IC : bool; PC : bool; TI : int)
returns(equivalence);
var mtl, ccs : bool;
let
  mtl = MTL(TR, IC, PC, TI);
  ccs = CoCoSim(TR, IC, PC, TI);
  equivalence = mtl=ccs;
tel
```

Listing 7.2: Lustre code for equivalence node

7.3 Results

We tried to verify the equivalence node for each of the formalization types using Zustre, which returns whether the CoCoSim and MTL nodes are equivalent. The result for all four formalization types is `SAFE`, which means that the CoCoSim and MTL formalizations are equivalent. Thus, the formalization of the CEL template is consistent.

However, the manual encoding of the MTL templates into Lustre nodes was done by the same person who created the CoCoSim formalization. So, there is a risk that there is actually a difference between the MTL and CoCoSim formalizations, but that the MTL encoding is biased towards the corresponding CoCoSim formalization and thus contains the same mistakes. As will be discussed in Chapter 9, the consistency check should be automated. This involves the development of a parser that will encode MTL templates into Lustre nodes. This should preferably be done by a person with no knowledge of the CoCoSim formalization, to ensure an unbiased comparison.

Part IV

Conclusion

Chapter 8

Conclusion

The purpose of this research is to evaluate the challenges of developing requirements in real projects, and to develop automated support for requirements elicitation. For this purpose, we have extended the FRET environment and have also connected it to the CoCoSim tool by defining a formalization of the CEL template into a CoCoSim template. Moreover, we have carried out two major case studies, comparing the current situation of specifying requirements in natural language and then manually formalizing them to the situation in which FRET is used to specify and automatically formalize requirements.

The FRET framework should provide an environment for the specification of unambiguous requirements, automated formalization and connection to analysis tools, and high configurability.

This report described how the existing functionality of using requirement templates in FRET could be used to support the unambiguous specification of requirements and how this could be combined with formalization into a CoCoSim format.

This has been demonstrated using the CEL template. This report also discussed the formalization of the CEL template into the CoCoSim format (and MTL), and how a Simulink model for a requirement can be created directly from within FRET. Finally, it discussed how subject matter experts can directly define their own (domain pattern) templates, parsers, and formalizations in FRET, in order to offer a large degree of configurability to its users.

For the development of the FRET framework, we needed to answer two research questions:

(1a) **How can requirement templates be used to specify unambiguous requirements without requiring significant background knowledge of any formalism?**

In Section 4.1, we discussed how FRET makes use of requirement templates that are used to specify requirements. We allow subject matter experts to create their own (natural language) templates and other resources, after which end users only need to instantiate the parameters fields of these templates to specify requirements. The subject matter expert is responsible for making the templates suitable and unambiguous, so the end users do not need to concern themselves with this or any formalisms they want to transform their requirements into.

This approach aids users in making their requirements unambiguous, but it does highly depend on the subject matter expert's capability to create suitable templates

(1b) **How can unambiguous requirements be formalized automatically into a chosen formalism?**

In Section 4.2, we discussed how FRET uses templates and parsers to formalize the previously specified unambiguous requirement into a formalism of the user's choice. This requires subject matter experts to first define formalization templates for the chosen requirement template and formalism. The end users do not need any background knowledge for this and can transform their requirements into any formalism that was specified by the subject matter expert with just a few clicks.

FRET was evaluated through three research questions:

1. **Performance**

   We showed that FRET was able to specify 100% of the 66 requirements that we could manually specify. Furthermore, of this set of requirements, we only managed to manually formalize 36 into CoCoSim requirements. We could also unambiguously specify 34 of these in FRET, which made it possible to formalize them through FRET. Thus, 94% of requirements that we could manually formalize, we could also formalize automatically using FRET.

2. **Usability**

   We determined FRET's usability by comparing it to the process of manually formalizing requirements. Our conclusion is that specification and formalization using FRET takes less time and effort overall, mostly because it automates formalization. Furthermore, FRET has the potential to be much less error-prone, but this highly depends on the way it is used. We attempt to maintain a balance between minimizing errors and allowing users' freedom in the way they want to specify their requirements. This means that it is entirely up to the subject matter expert, when specifying the templates, how much freedom the end users should get when instantiating these templates. The more freedom, the more risk of errors. Finally, the difficulty in using FRET is distributed differently compared to the difficulty in specifying and formalizing requirements manually. FRET is more difficult to use for subject matter experts, because they must learn how to use the configurable features of FRET. On the other hand, FRET is much easier to use by end users, because they no longer need to concern themselves with the formalization of requirements. Instead of requiring that someone formalizes each and every requirement into any formalism, FRET only requires an expert to formalize templates, after which anyone can use these formalizations without further knowledge of the syntax.

3. **Consistency**
   We proved that the CoCoSim formalization of the CEL template is equivalent to its MTL formalization. This shows that FRET's formalization capabilities are at least consistent for the CEL template. This same approach can be used for showing consistency for any other template. Chapter 9 will give suggestions for how we can make it easier for subject matter experts to establish consistency for other templates and for actual requirements in which the parameter fields are also instantiated.

Overall, we have shown how the concept of requirement templates can be used to automate the process of requirements formalization. Due to FRET's high configurability, this approach can actually be used to specify and automatically formalize a broad range of requirements.

Chapter 9

Future work

9.1 Usability

It is important that FRET is easy to use, because otherwise people will continue to manually specify and formalize requirements. However, this should be evaluated on a larger scale. Currently, FRET has only been used and tested within the FRET development team. Therefore, more case studies and real-life evaluations by non-experts should be performed to improve FRET further. This will first require research in the field of HMI, to help us create an initial user interface for FRET. We can later evaluate and improve this user interface through empirical research in the field of software engineering, by allowing users to use FRET and provide feedback. It would be interesting to obtain users' opinions on both how intuitive the user interface is and to obtain measurements on how much time and how many steps steps it requires to complete their actions.

9.2 Configurability

Subject matter experts can add, edit, and remove (domain pattern) templates, parsers, and formalizations. However, this currently requires that they edit the source files, which has a number of disadvantages:

- *Requires knowledge of syntax*
  Users must understand the syntax in which these configurables are specified and be capable of creating this by themselves. This requires an investment to become familiar with FRET's internal structure.
- *There is no control over these changes*
  This means that the existing configurables could easily be corrupted.

In the future, we would like to make the configurability fully independent of FRET's implementation, so that users can configure FRET from within the program and not through source files.

9.3 Consistency

The only guarantee users currently have is that the CoCoSim and MTL formalizations of the CEL template are consistent. This poses a number of questions:

- Are formalizations of requirements based on the CEL template also consistent?
  Only the template is checked for equivalence, but not the parsers. In order to make sure that formalizations of requirements based on the CEL template are also equivalent, we should also determine the consistency of the CoCoSim and MTL field parsers. Otherwise, for example, we might create the following requirement by instantiating the CEL template:

  > Whenever true and input==1, then output==1 must become true within 1 frames.

  We have already established that the formalization of this template into MTL and CoCoSim is consistent. However, we do not know what happens to the parameter fields. For example, IC is in this case equal to 'input==1', which is parsed by the Condition parser. However, the MTL Condition transformer might transform this into an MTL formula that is equivalent to 'input==1', whereas the CoCoSim transformer might transform it into a CoCoSim requirement that is equivalent to 'input==2'. These transformers are currently not verified in any way, so we cannot say for certain that they always transform correctly.

- Are custom templates consistent?
  One of FRET's strength is the capability for adding your own templates, in case the CEL template is not sufficient to specify your requirements. However, we currently do not supply a way to check the consistency of other templates. The consistency of the CEL template was determined by manually translating the MTL template into Lustre and manually creating a Lustre node to compare the auto-generated CoCoSim Lustre node to the MTL Lustre node. However, we should attempt to automate this process, so that it can be used for any template. This requires that a compiler is developed to compile MTL into Lustre, the equivalence node must be created automatically the Lustre nodes must be combined and checked automatically. Furthermore, we should provide a way for users to add their own compilers from other formalizations to Lustre, or other intermediate formats.

9.4 Testing

Because we only check for consistency, it is still possible that a user specifies a requirement that means something else than the user realizes. We want to introduce a form of testing to reduce this risk. If a user creates a template (or requirement), we want to show a number of auto-generated test cases, so that the user can easily see if the template performs as expected. Also, users should be able to add their own test cases. Through this form of testing, users can make sure that they cover all the edge cases and do not have some hidden ambiguities in their templates.

For the auto-generated test cases, we will have to investigate the field of scenario-based requirements engineering. This will help us create suitable scenario's to give users a better intuition of the meaning of a requirement. Allowing users to create their own test cases for requirements requires research in the field of executable specifications: the requirement specifications will need to be executed to allow users to test their own sets of inputs and outputs.

Appendix A

Lockheed Martin challenges

This appendix contains brief descriptions of the models and their requirements as provided by Lockheed Martin.

A.1 Challenge 0

This challenge problem involves the verification of a redundancy management system using quantum simulation techniques. The purpose of this redundancy management system is to prevent errors from propagating past the input portion of an airborne application. The assumed platform configuration is a set of three computers which execute identical software applications, and which each take the same set of input values from sensors on the aircraft. For each set of triplex inputs, the redundancy management system must monitor for failures (or imminent failures), report the failure status of the set to the necessary functions, and choose an acceptable signal for computational use. All of the inputs used in redundancy management are triply redundant and have some type of tie-breaking capability.

A.1.1 Requirements

1. In the no-fail state, a miscompare, which shall be characterized by one branch differing with the other two branches by a unique trip level that lasts for more than the persistence limit, shall be reported to failure management as a failure.
2. In the no-fail state, the mid-value shall be the selected value. Note: a first failure in progress will not affect the method for determining the selected value.

3. In the single fail state, a good channel average of the remaining two good branches shall be used to determine the selected value.
4. If a second failure is in progress, the selected value shall remain unchanged from the previous selected value.

A.1.2 Discussion

Requirement 4 states what should happen when a second failure is in progress. This means that the system is in the single fail state and a failure is currently in progress. However, requirement 3 states what should happen if the system is in the single fail state. These two requirements overlap, so we update requirement 3 to state what should happen if the system is in the single fail state *without* a failure in progress:

3. In the single fail state, if a second failure is not in progress, a good channel average of the remaining two good branches shall be used to determine the selected value.

A.2 Challenge 1

This model represents an abstraction of detailed design requirements for an Advanced Autopilot System interacting with an independent sensor platform. The Finite State Machine (FSM) represents a cyber-physical system with two independent components executing in real time for the purpose of ensuring a safe automatic operation in the vicinity of hazardous obstacles. The autopilot system, tightly integrated with the vehicle flight control computer, is responsible for commanding a safety maneuver in the event of a hazard. The sensor is the reporting agent to the autopilot with observability on imminent danger. When the conditions are met to allow an autopilot operation (supported and no failures) and the pilot is not manually controlling the vehicle (standby is false), the autopilot will behave normally until a maneuver is required in the event of a hazard (sensor is not good).

A.2.1 Requirements

1. Exceeding sensor limits shall latch an autopilot pullup when the pilot is not in control (not standby) and the system is supported without failures (not apfail).
2. The autopilot shall change states from TRANSITION to STANDBY when the pilot is in control (standby).

3. The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.
4. The autopilot shall change states from NOMINAL to MANEUVER when the sensor data is not good.
5. The autopilot shall change states from NOMINAL to STANDBY when the pilot is in control (standby).
6. The autopilot shall change states from MANEUVER to STANDBY when the pilot is in control (standby) and sensor data is good.
7. The autopilot shall change states from MANEUVER to TRANSITION when the system is supported and sensor data is good.
8. The autopilot shall change states from STANDBY to TRANSITION when the pilot is not in control (not standby).
9. The autopilot shall change states from STANDBY to MANEUVER when a failure occurs (apfail).
10. The sensor shall change states from NOMINAL to FAULT when limits are exceeded.
11. The sensor shall change states from NOMINAL to TRANSITION when the autopilot is not requesting support (not request).
12. The sensor shall change states from FAULT to TRANSITION when the autopilot is not requesting support (not request) and limits are not exceeded (not limits).
13. The sensor shall change states from TRANSITION to NOMINAL when the autopilot is requesting support (request) and the autopilot reports the correct active mode (mode).

A.2.2 Discussion

Requirement 2 states what should happen when the autopilot is in state TRANSITION and standby is true, whereas requirement 3 states what should happen when the autopilot is in state TRANSITION, supported is true and good is true. These overlap when the autopilot is in state TRANSITION, standby is true, supported is true and good is true. Similar situations occurs for requirements 4 and 5, 6 and 7, 8 and 9, and 10 and 11.

Based on the description of the system, we update the following requirements:

3. The autopilot shall change states from TRANSITION to NOMINAL when the pilot is not in control, there are no failures, the system is supported and sensor data is good.

4. The autopilot shall change states from NOMINAL to MANEUVER when the sensor data is not good and the pilot is not in control.

7. The autopilot shall change states from MANEUVER to TRANSITION when the system is supported and sensor data is good and the pilot is not in control.

9. The autopilot shall change states from STANDBY to MANEUVER when a failure occurs (apfail) and the pilot is not in control.

11. The sensor shall change states from NOMINAL to TRANSITION when the autopilot is not requesting support (not request) and limits are not exceeded.

A.3 Challenge 2

A common flight control utility for computing the Tustin Integration of a signal $yout = T/2 * (xin + xinpv) + ypv$. The algorithm bounds the allowable integration range with a position limiter, with TL as the Top Limit, and BL as the Bottom Limit. Other inputs are the signal to be integrated (xin), the time step (dt), a Boolean reset flag (reset), and the initial condition upon a reset condition (ic). A provision has been implemented for the limiter functionality in the algorithm. If the user plumbs a TL value that is less than BL, the algorithm will swap these numerical values to correctly bound the signal.

A.3.1 Requirements

1. When Reset is True and the Initial Condition (ic) is bounded by the provided Top and Bottom Limits ($BL <= ic <= TL$), the Output (yout) shall equal the Initial Condition (ic).

2. The Output (yout) shall be bounded by the provided Top and Bottom limits (TL and BL).

3. When in normal operation, the output shall be the result of the equation, $yout = T/2 * (xin + xinpv) + ypv$.

4. The Output of this function shall approximate the integration of the value of the input signal over time within a specified tolerance, defined in subtests below:

(a) After 10 seconds of Computation at an execution frequency of 10 hz, the Output should equal 10 within a +/- 0.1 tolerance, for a Constant Input ($xin = 1.0$), and the sample delta time $T = 0.1$ seconds when in normal mode of operation.

(b) Over a 10 second computational duration at an execution frequency of 10 hz, the Output should equal the sine of time t, sin(t), where time is defined as a vector from 0 to 10 by increments of 0.1 seconds within a +/- 0.1 tolerance for an input equal to the cosine of time t, cos(t), with the sample delta time $T = 0.1$ seconds when in normal mode of operation.

A.3.2 Discussion

The timing requirements of requirement 4 are a bit unclear, so we assume that part (a) refers to when the computation is at <u>exactly</u> 10 seconds, not any time after that and we assume that (b) refers to the computation until and <u>not including</u> 10 seconds.

A.4 Challenge 3

This example model demonstrates a simplified regulators inner loop architecture used in many feedback control applications. The model includes two subsystems. The first is the input subsystem which is used solely for signal routing with variable renaming and bus creation. The second subsystem is the primary system under test, the regulators algorithm. The regulators algorithm consists of 5 classical controllers for establishing the desired dynamics of a vehicle for the roll, pitch, yaw, axial, and height channels.

A.4.1 Requirements

1. The Inner Loop Roll Regulator Shall not command angular roll accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

2. The Inner Loop Pitch Regulator Shall not command angular pitch accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

3. The Inner Loop Yaw Regulator Shall not command angular yaw accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

4. The Inner Loop Airspeed Regulator Shall not command translational axial accelerations greater than the capability of the system (32 ft/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

5. The Inner Loop Height Regulator Shall not command translational height accelerations greater than the capability of the system (32 ft/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

6. The Inner Loop Roll Regulator Shall not command transient changes in angular roll acceleration greater than 50 deg/sec2/sec.

7. The Inner Loop Pitch Regulator Shall not command transient changes in angular pitch acceleration greater than 50 deg/sec2/sec.

8. The Inner Loop Yaw Regulator Shall not command transient changes in angular yaw acceleration greater than 50 deg/sec2/sec.

9. The Inner Loop Airspeed Regulator Shall not command transient changes in translational axial acceleration greater than 32 ft/sec2/sec.

10. The Inner Loop Height Regulator Shall not command transient changes in translational height acceleration greater than 32 ft/sec2/sec.

A.5 Challenge 4

This example is a nonlinear algorithm for generating a guidance command for an air vehicle. The vector method determines an intercept location in Euclidean space for a target (static) referred to as aim points. The intercept location is a function of the relative position between the vehicle and target, where a minimum relative position to maintain is specified as a desired standoff distance.

A.5.1 Requirements

1. NLGuidance shall always maintain the target on the port-side of the vehicle.

2. NLGuidance shall compute the inertial position vector for aim point 1, defining the location at Standoff 1 with an offset from the target position, and oriented on a vector perpendicular to the tangent relative position vector from the vehicle to the corresponding aim point.

3. NLGuidance shall compute the inertial position vector for aim point 2, defining the location at Standoff 2 with an offset from the target position, and oriented on

a vector perpendicular to the tangent relative position vector from the vehicle to the corresponding aim point.

4. NLGuidance shall always select an inertial position vector of aim point 1 or 2 which shall result in a counter clockwise loiter for the UAV. For example, the picture above with vehicle position, UAV, would return Tangent Aim Point 2.

5. When the UAV relative position to the target is less than the minimum standoff distance, NLGuidance shall command the nearest inertial position in order to reestablish the minimum standoff distance while maintaining the target on the port-side of the vehicle.

6. NLGuidance shall output consistent aim point with a static target without appreciable transient behavior in the command generation other than aim point switching where a transient is required to maintain a counter clockwise loiter (ref requirement 3). Appreciable transient behavior is defined as erratic changes in the aim point command, beyond the following specific tolerance:

   (a) The change in the magnitude of the output over one frame of execution with T sample period shall not exceed the quantity of the combined velocity of the target plus the velocity of the vehicle multiplied by T.

7. NLGuidance shall output the equivalent altitude of the vehicle for in-plane navigation. In-plane navigation is defined where the target and the vehicle altitude (3 rd component in the input inertial position vectors) are equal.

A.5.2 Discussion

The description assumes a lot of background knowledge. Due to a lack of this knowledge, we do not know what the requirements mean in terms of the model's inputs and outputs.

A.6 Challenge 5

This example is a two-input single-output two hidden layer feed forward nonlinear 2x10x10x1 neural network. Neural networks of this form are common utilities in modeling and simulation for capturing complex numerical dependencies. In this example, a single dependent variable, z, is computed based on two independent parameters, x and y.

A.6.1 Requirements

1. The maximum value of the NN output, z, shall always be less than or equal to 1.1, regardless of the input values.
2. The minimum value of the NN output, z, shall always be greater than or equal to -0.2, regardless of the input values.
3. Using a first order finite backward difference equation, the spatial derivatives of $\Delta z/\Delta xt = (z(n,1) - z(n-1))/(xt(n,1) - xt(n-1,1)$ and $\Delta z/\Delta yt = (z(n,1) - z(n-1))/(yt(n,1) - yt(n-1,1)$ shall never exceed a top bound of +10 or bottom bound of -35 (e.g. $-35 <= \Delta z/\Delta(xt, yt) <= 10$), where n denotes an index to the current values and n-1 denotes the prior values in the included truth data for xt and yt.
4. The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, for the equivalent input of (xt, yt).

A.7 Challenge 6

This example provides a subset of an algorithm commonly referred to as the control allocation method, which enables the calculation of the optimal effector (surface) configuration for a vehicle, given a problem type (typically desired acceleration error, or desired control minimization effort, or a combination of both). In this specific case, the problem type is control minimization of the form:

Minimize

$$J(\Delta u) = (\Delta u - \Delta up)^T * Wp * (\Delta u - \Delta up)$$

Subject to

$$B * \Delta u = \Delta d$$

where J represents the cost of the control effort, u is the control solution, up is the preferred control solution, Wp is a Weighting matrix, B is the linearized control effectivity matrix, and d is the desired acceleration error [43]. In this specific case for the over-determined solution, when there are more surface effectors than commanded axes (n¡m) where n is the length of the d vector, and m is the length of the u vector, an analytic solution exists [43].

$$\Delta u = P\&\Delta d$$

where

$$P = (Wp^T)^{-1} * B^T(B * (Wp^T)^{-1} * B^T)^{-1}$$

A.7.1 Requirements

1. When the determinant of B(inv(Wp)B is $<= 1e-12$ as indicated by ridge_on set to True, the inversion of the B(inv(Wp)B matrix should be accurate to 6 digits precision in that each element in the check output matrix is within a 1e-6 tolerance with respect to the elements of a 3x3 identity matrix.
2. When the determinant of B(inv(Wp)B is $> 1e-12$ as indicated by ridge_on set to False, the inversion of the B(inv(Wp)B matrix should be accurate to 12 digits precision in that each element in the check output matrix is within a 1e-12 tolerance with respect to the elements of a 3x3 identity matrix.
3. The output u vector should be a 5x1 vector.
4. The 2-norm of the output Buminusd should be less than 0.01.

A.7.2 Discussion

The Simulink model for this challenge makes heavy use of matrix computations. These blocks are part of some matrix library that was not provided with the model. Because of this, the model is incomplete. Furthermore, the requirements would also use need this library. Therefore, we could not use this challenge for our research.

A.8 Challenge 7

This example includes a full six degree of freedom simulation of the DeHavilland Beaver airplane with autopilot. This system represents a realistic model environment where formal methods analysis could prove to be extremely beneficial to help the designer prove aspects of the closed loop system without exhaustive Monte-carlo Simulation.

A.8.1 Requirements

1. The altitude hold autopilot shall maintain altitude within 35 feet of the initial condition.

2. Roll Autopilot shall engage when the pilot selects the autopilot engage switch in the cockpit and disengage when the switch is deselected. When not engaged, the command to the roll actuator shall be zero.
3. Roll hold mode shall be the active mode whenever the autopilot is engaged and no other lateral mode is active.
4. When roll hold mode becomes the active mode the roll hold reference shall be set to the actual roll attitude of the aircraft, except under the following conditions: The roll hold reference shall be set to zero if the actual roll angle is less than 6 degrees, in either direction, at the time of roll hold engagement. The roll hold reference shall be set to 30 degrees in the same direction as the actual roll angle if the actual roll angle is greater than 30 degrees at the time of roll hold engagement. The roll reference shall be set to the cockpit turn knob command, up to a 30 degree limit, if the turn knob is commanding 3 degrees or more in either direction.
5. Steady state roll commands shall be tracked within 1 degree in calm air. Response to roll step commands shall not exceed 10in calm air. Small signal (<3 degree) roll bandwidth shall be at least 0.5 rad/sec.
6. The maximum roll rate for large commands shall be 6 deg/sec +/- 10% in calm air.
7. The maximum roll angle allowed shall be 30 deg +/-10% in calm air.
8. The maximum aileron command allowed shall be 15 deg.
9. Heading Hold shall become the active mode when the pilot selects the heading switch in the cockpit and deactivate when the switch is deselected.
10. When heading hold mode becomes the active mode the heading hold reference shall be set by the pilot via a cockpit control.
11. Steady state heading commands shall be tracked within 1 degree in calm air. Response to heading step commands shall not exceed 10overshoot in calm air.

A.8.2 Discussion

The description assumes a lot of background knowledge. Due to a lack of this knowledge, we do not know what the requirements mean in terms of the model's inputs and outputs.

A.9 Challenge 8

This example provides a safety algorithm for monitoring airspeed in the SWIM (System Wide Integrity Monitor) suite in order to provide warning to an operator when the vehicle speed is approaching a boundary where an evasive flyup maneuver cannot be achieved.

A.9.1 Requirements

1. The SWIM Airspeed algorithm shall output the minimum AGCAS airspeed required to perform a 2g flyup as follows:

   Auto GCAS Minimum Vcas (knots) =

   $SQRT\{[2 *$ Load Factor $*$ Gross Weight$]/[$Air Density at sea level$]*$ Coefficient of Lift Max $*$ Wing Area$]\}/1.6891$(ft/s/knots)

   where

   Load Factor = 2 gs

   Gross weight of the aircraft

   Air Density at sea level = 0.0023769slugs/ft^3

   Coefficient of Lift Max = 1.24(CAT I) and 1.10 (CAT III)

   Wing Area of the F-16 = 300 ft^2

   Thus,

   - IF CAT I, Auto GCAS Minimum Vcas (knots) = $1.25921 * \sqrt{\text{(Gross Weight)}}$ +10.0
   - IF CAT III, Auto GCAS Minimum Vcas (knots) = $1.33694 * \sqrt{\text{(Gross Weight)}}$ +10.0

2. When a low speed warning is allowed, as computed by the SWIM Airspeed algorithm, a low speed warning shall be true when the vehicle air data impact pressure is less than the warning trigger for minimum impact pressure in which a safe AGCAS evasive maneuver can be accomplished where:

The warning trigger for minimum impact pressure in which a safe AGCAS evasive maneuver can be accomplished is computed as

$$\begin{aligned}\text{SWIM\_Qcmin\_lbspft2} &= [(-2.0906 + 0.020306 * \text{Auto GCAS Minimum Vcas}) \\ &\quad + 0.1] * (70.7184(\text{lbspft2/in Hg}))\end{aligned}$$

A.10 Challenge 9

This component creates a Rotation Matrix describing a 321 rotation about the z-axis, y-axis, and finally x-axis of an Inertial frame in Euclidean space, and, given an Input Vector, outputs the representation of the Input Vector in the new rotated frame. Note the y-axis and x-axis subsequent rotations are the second and third transform operations, identified in the 321 operation by the 2 and 3 label description, respectively. The latter two operations are rotations about the intermediate frames. The computed Rotation Matrix Shall Equal a Special Orthogonal(3) 3x3 with the following properties: (i) the rows and columns of the Output Matrix shall be orthonormal; (ii) the Output Matrix multiplied with the Transpose of the Output Matrix shall be the Identity (3x3) Matrix; and (iii) the determinant of the Output Matrix shall be equal to 1.

A.10.1 Requirements

1. The Rotation Matrix Output, DCM321, of this Function Shall Equal a 3x3 Matrix Product of a 3x3 Euler 3 (Roll) Rotation Matrix times a 3x3 Euler 2 (Pitch) Rotation Matrix times a 3x3 Euler 1 (Heading) Rotation Matrix.

2. The Body Vector Output, Vb, of this Function Shall Equal a 3x1 Vector Product of the 3x3 Rotation Matrix Output, DCM321, times the Input Inertial Vector, Vi.

3. The magnitude of the Body Vector Output, Vb, shall equal the magnitude of the Input Inertial Vector, Vi.

4. The Rotation Matrix, DCM321, shall be invertible with the exception of the case where theta = +/- pi/2 radians.

5. The Rotation Matrix, DCM321, shall provide a distinct mapping from the input vector, Vi, to the output vector, Vb, for each pitch angle, theta. Note: the DCM321 is not distinct for all phi and psi inputs.

6. The rows and columns of the Rotation Matrix, DCM321, shall be orthonormal. For instance, denoting r1 as row 1 and r2 as row 2 of DCM321, $< r1, r2 >= r1$

$r2T = 0$ and $< r1, r1 >= r1\ r1T = 1$. Likewise, with c1 as column 1 and c2 as column 2 of DCM321, $< c1, c2 >= c1\ c2T = 0$ and $< c1, c1 >= c1\ c1T = 1$.

7. The Rotation Matrix, DCM321, multiplied by the transpose of the Rotation Matrix, DCM321T shall be the Identity (3x3) Matrix.

8. The determinant of the Rotation Matrix, |DCM321|, shall be equal to 1.0.

A.10.2 Discussion

The Simulink model for this challenge makes heavy use of matrix computations. These blocks are part of some matrix library that was not provided with the model. Because of this, the model is incomplete. Furthermore, the requirements would also use need this library. Therefore, we could not use this challenge for our research.

Appendix B

Templates

This appendix contains all the (domain pattern) templates that were created and used to specify and formalize the LM challenges. These templates are not fixed; different subject matter experts might make different choices to better suit their team.

B.1 Global templates

Name	Conditional Event Logic		
Fields	Whenever		
	<Co[Trigger]:TR>		
	and		
	<Co[InitialConditions]:IC>		
	, then		
	<Co[PostConditions]:PC>		
	must become true within		
	<Co[Time]:TI>		
	.		
Questions	Q1: If a new Trigger occurs, does it release the obligations set by the previous Trigger?		`<Boolean>`
	Q2: Do the InitialConditions need to be true only at the time where the Trigger is true?		`<Boolean>`
Field types	TR	<Any>	
		true	
		<Parser:Condition>	
	IC	<Any>	
		true	
		<Parser:Condition>	
	PC	<Any>	
		<Parser:Condition>	
	TI	<Any>	
		1 frames	
		<Parser:Duration>	

Name	Free Form	
Fields	<Co[Enter any text]:FF>	
	.	
Field types	FF	<Any>

B.2 Challenge 0

Report Failure

Name	Report Failure	
Fields	In the no-fail state, if there is a miscompare of	
	<Co[Branch1]:B1>	
	with	
	<Co[Branch2]:B2>	
	and	
	<Co[Branch3]:B3>	
	, the failure state shall be set to	
	<Co[FaultCode]:FC>	
	.	
Field types	B1	ia
		ib
		ic
	B2	ia
		ib
		ic
	B3	ia
		ib
		ic
	FC	1
		2
		4
Target template	Conditional Event Logic	
Mapping	TR	true
	IC	previous(FC)==0 && |[B1]-[B2]| >Tlevel && |[B1]-[B3]| >Tlevel && PC>PClimit
	PC	FC=[FC]
	TI	1 frames

Select Value

Name	Select Value	
Fields	If the failure state is	
	<Co[FaultCode]:FC>	
	and	
	<Co[InitialConditions]:IC>	
	, then the selected value will be set to	
	<Co[SelectedValue]:SV>	
	.	
Field types	FC	0
		1
		2
		4
	IC	<Any>
		true
		<Parser:Condition>
	SV	<Parser:Arithmetic>
Target template	Conditional Event Logic	
Mapping	TR	true
	IC	previous(FC)==[FC] && [IC]
	PC	sel_val==[SV]
	TI	1 frames

B.3 Challenge 1

State Change

Name	State Change	
Fields	The module	
	<Co[Module]:M>	
	shall change states from	
	<Co[PrevState]:PS>	
	to	
	<Co[NextState]:NS>	
	whenever	
	<Co[Conditions]:C>	
	.	
Field types	M	<Any>
	PS	<Int>
	NS	<Int>
	C	<Parser:Condition>
Target template	Conditional Event Logic	
Mapping	TR	true
	IC	state==[PS] and [C]
	PC	STATE==[NS]
	TI	1 frames

Autopilot State Change

Name	Autopilot State Change
Fields	The autopilot shall change states from
	<Co[PrevState]:PS>
	to
	<Co[NextState]:NS>
	whenever standby is
	<Co[Standby]:ST>
	and limits is
	<Co[Limits]:LI>
	and supported is
	<Co[Supported]:SU>
	and apfail is
	<Co[Apfail]:AP>
	and good is
	<Co[Good]:GO>
	.

Field types	PS	0
		1
		2
		3
	NS	0
		1
		2
		3
	ST	true
		false
		standby
	LI	true
		false
		limits
	SU	true
		false
		supported
	AP	true
		false
		apfail
	GO	true
		false
		good
Target template	State Change	
Mapping	M	autopilot
	PS	[PS]
	NS	[NS]
	C	standby==[ST] && limits==[LI] && supported==[SU] && apfail==[AP] && good==[GO]

Sensor State Change

Name	Sensor State Change	
Fields	The sensor	
	shall change states from	
	<Co[PrevState]:PS>	
	to	
	<Co[NextState]:NS>	
	whenever limits is	
	<Co[Limits]:LI>	
	and request is	
	<Co[Request]:RE>	
	and mode is	
	<Co[Mode]:MO>	
	.	
Field types	PS	0
		1
		2
	NS	0
		1
		2
	LI	true
		false
		limits
	RE	true
		false
		request
	MO	true
		false
		mode
Target template	State Change	
Mapping	M	autopilot
	PS	[PS]
	NS	[NS]
	C	limits==[LI] && request==[RE] && mode==[MO]

B.4 Challenge 2

No templates were created for this challenge.

B.5 Challenge 3

Maximum Acceleration

Name	Maximum Acceleration	
Fields	The regulator	
	<Co[Regulator]:R>	
	shall not command accelerations greater than	
	<Co[SystemCapability]:SC>	
	<Co[Unit]:U>	
	for durations exceeding 100 frames.	
Field types	R	lcvdt_cmd_fcs_dps2
		mcvdt_cmd_fcs_dps2
		ncvdt_cmd_fcs_dps2
		xcvdt_cmd_fcs_dps2
		hcvdt_cmd_fcs_dps2
	SC	50
		32
	U	deg/sec^2
		ft/sec^2
Target template	Conditional Event Logic	
Mapping	TR	true
	IC	true
	PC	[R]¡=[SC]
	TI	100 frames

Transient Change

Name	Transient Change	
Fields	The regulator	
	<Co[Regulator]:R>	
	shall not command changes in acceleration greater than	
	<Co[SystemCapability]:SC>	
	<Co[Unit]:U>	
	.	
Field types	R	lcvdt_cmd_fcs_dps2
		mcvdt_cmd_fcs_dps2
		ncvdt_cmd_fcs_dps2
		xcvdt_cmd_fcs_dps2
		hcvdt_cmd_fcs_dps2
	SC	50
		32
	U	deg/sec^2/sec
		ft/sec^2/sec
Target template	Conditional Event Logic	
Mapping	TR	true
	IC	true
	PC	—[R]-previous([R])—¡=0.5
	TI	1 frames

B.6 Challenge 4

Because the requirements of this challenge could not be formalize, no templates were created.

B.7 Challenge 5

Output Value

Name	Output Value	
Fields	The maximum value of the NN output, z, shall always be	
	<Co[RelationalOperator]:RO>	
	<Co[Value]:V>	
	, regardless of the input values.	
Field types	RO	<=
		>=
	V	<Int>
Target template	Conditional Event Logic	
Mapping	TR	true
	IC	true
	PC	z [RO] [V]
	TI	1 frames

B.8 Challenge 6

Because the requirements of this challenge could not be formalize, no templates were created.

B.9 Challenge 7

Because the requirements of this challenge could not be formalize, no templates were created.

B.10 Challenge 8

No templates were created for this challenge.

B.11 Challenge 9

Because the requirements of this challenge could not be formalize, no templates were created.

Appendix C

Requirements

This appendix contains all the requirements of the LM challenges, as instantiated in FRET.

C.1 Challenge 0

Requirement 1

In the no-fail state, a miscompare, which shall be characterized by one branch differing with the other two branches by a unique trip level that lasts for more than the persistence limit, shall be reported to failure management as a failure.

Id	1	
Template	Free Form	
Fields	FF	In the no-fail state, a miscompare, which shall be characterized by one branch differing with the other two branches by a unique trip level that lasts for more than the persistence limit, shall be reported to failure management as a failure.

Id	1.1	
Template	Free Form	
Fields	FF	In the no-fail state, a miscompare of branch A shall be reported to failure management as a failure.

Id	1.1.1	
Template	Report Failure	
Fields	B1	ia
	B2	ib
	B3	ic
	FC	4

Id	1.2	
Template	Free Form	
Fields	FF	In the no-fail state, a miscompare of branch B shall be reported to failure management as a failure.

Id	1.2.1	
Template	Report Failure	
Fields	B1	ib
	B2	ia
	B3	ic
	FC	2

Id	1.3	
Template	Free Form	
Fields	FF	In the no-fail state, a miscompare of branch C shall be reported to failure management as a failure.

Id	1.3.1	
Template	Report Failure	
Fields	B1	ic
	B2	ia
	B3	ib
	FC	1

Requirement 2

In the no-fail state, the mid-value shall be the selected value. Note: a first failure in progress will not affect the method for determining the selected value.

Id	2	
Template	Free Form	
Fields	FF	In the no-fail state, the mid-value shall be the selected value. Note: a first failure in progress will not affect the method for determining the selected value.

Id	2.1	
Template	Select Value	
Fields	FC	0
	IC	true
	SV	max(min(a,b),min(max(a,b),c))

Requirement 3

In the single fail state, a good channel average of the remaining two good branches shall be used to determine the selected value.

Id	3	
Template	Free Form	
Fields	FF	In the single fail state, a good channel average of the remaining two good branches shall be used to determine the selected value.

Id	3.1	
Template	Free Form	
Fields	FF	If branch A fails, the selected value is the good channel average of branches B and C.

Id	3.1.1	
Template	Select Value	
Fields	FC	4
	IC	|ib-ic| <=Tlevel
	SV	|ib+ic|/2

Id	3.2	
Template	Free Form	
Fields	FF	If branch B fails, the selected value is the good channel average of branches A and C.

Id	3.2.1	
Template	Select Value	
Fields	FC	2
	IC	|ia-ic| <=Tlevel
	SV	|ia+ic|/2

Id	3.3	
Template	Free Form	
Fields	FF	If branch C fails, the selected value is the good channel average of branches A and B.

Id	3.3.1	
Template	Select Value	
Fields	FC	1
	IC	|ia-ib| <=Tlevel
	SV	|ia+ib|/2

Requirement 4

If a second failure is in progress, the selected value shall remain unchanged from the previous selected value.

Id	4	
Template	Free Form	
Fields	FF	If a second failure is in progress, the selected value shall remain unchanged from the previous selected value.

Id	4.1	
Template	Free Form	
Fields	FF	If branch A fails and a second failure is in progress, the selected value is the previous selected value.

Id	4.1.1	
Template	Select Value	
Fields	FC	4
	IC	|ib-ic| >Tlevel
	SV	previous(sel_val)

Id	4.2	
Template	Free Form	
Fields	FF	If branch B fails and a second failure is in progress, the selected value is the previous selected value.

Id	4.2.1	
Template	Select Value	
Fields	FC	2
	IC	|ia-ic| >Tlevel
	SV	previous(sel_val)

Id	4.3	
Template	Free Form	
Fields	FF	If branch C fails and a second failure is in progress, the selected value is the previous selected value.

Id	4.3.1	
Template	Select Value	
Fields	FC	1
	IC	|ia-ib| >Tlevel
	SV	previous(sel_val)

C.2 Challenge 1

Requirement 1

Exceeding sensor limits shall latch an autopilot pullup when the pilot is not in control (not standby) and the system is supported without failures (not apfail).

Id	1	
Template	Free Form	
Fields	FF	Exceeding sensor limits shall latch an autopilot pullup when the pilot is not in control (not standby) and the system is supported without failures (not apfail).

Id	1.1	
Template	Conditional Event Logic	
Fields	TR	limits==true
	IC	standby==false && supported==true && apfail==false
	PC	pullup==true
	TI	1 frames

Requirement 2

The autopilot shall change states from TRANSITION to STANDBY when the pilot is in control (standby).

Id	2	
Template	Free Form	
Fields	FF	The autopilot shall change states from TRANSITION to STANDBY when the pilot is in control.

Id	2.1	
Template	Autopilot State Change	
Fields	PS	0
	NS	3
	ST	true
	LI	limits
	SU	supported
	AP	apfail
	GO	good

Requirement 3

The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.

Id	3	
Template	Free Form	
Fields	FF	The autopilot shall change states from TRANSITION to NOMINAL when the system is supported and sensor data is good.

Id	3.1	
Template	Autopilot State Change	
Fields	PS	0
	NS	1
	ST	false
	LI	limits
	SU	true
	AP	apfail
	GO	true

Requirement 4

The autopilot shall change states from NOMINAL to MANEUVER when the sensor data is not good.

Id	4	
Template	Free Form	
Fields	FF	The autopilot shall change states from NOMINAL to MANEUVER when the sensor data is not good.

Id	4.1	
Template	Autopilot State Change	
Fields	PS	1
	NS	2
	ST	standby
	LI	limits
	SU	supported
	AP	apfail
	GO	false

Requirement 5

The autopilot shall change states from NOMINAL to STANDBY when the pilot is in control (standby).

Id	5	
Template	Free Form	
Fields	FF	The autopilot shall change states from NOMINAL to STANDBY when the pilot is in control (standby).

Id	5.1	
Template	Autopilot State Change	
Fields	PS	1
	NS	3
	ST	true
	LI	limits
	SU	supported
	AP	apfail
	GO	true

Requirement 6

The autopilot shall change states from MANEUVER to STANDBY when the pilot is in control (standby) and sensor data is good.

Id	6	
Template	Free Form	
Fields	FF	The autopilot shall change states from MANEUVER to STANDBY when the pilot is in control (standby) and sensor data is good.

Id	6.1	
Template	Autopilot State Change	
Fields	PS	2
	NS	3
	ST	true
	LI	limits
	SU	supported
	AP	apfail
	GO	true

Requirement 7

The autopilot shall change states from MANEUVER to TRANSITION when the system is supported and sensor data is good.

Id	7	
Template	Free Form	
Fields	FF	The autopilot shall change states from MANEUVER to TRANSITION when the system is supported and sensor data is good.

Id	7.1	
Template	Autopilot State Change	
Fields	PS	2
	NS	0
	ST	false
	LI	limits
	SU	true
	AP	apfail
	GO	true

Requirement 8

The autopilot shall change states from STANDBY to TRANSITION when the pilot is not in control (not standby).

Id	8	
Template	Free Form	
Fields	FF	The autopilot shall change states from STANDBY to TRANSITION when the pilot is not in control (not standby).

Id	8.1	
Template	Autopilot State Change	
Fields	PS	3
	NS	0
	ST	false
	LI	limits
	SU	supported
	AP	apfail
	GO	good

Requirement 9

Id	9	
Template	Free Form	
Fields	FF	The autopilot shall change states from STANDBY to MANEUVER when a failure occurs (apfail).

Id	9.1	
Template	Autopilot State Change	
Fields	PS	3
	NS	2
	ST	true
	LI	limits
	SU	supported
	AP	true
	GO	good

Requirement 10

The sensor shall change states from NOMINAL to FAULT when limits are exceeded.

Id	10	
Template	Free Form	
Fields	FF	The sensor shall change states from NOMINAL to FAULT when limits are exceeded.

Id	10.1	
Template	Sensor State Change	
Fields	PS	0
	NS	2
	RE	requested
	LI	true
	MO	mode

Requirement 11

The sensor shall change states from NOMINAL to TRANSITION when the autopilot is not requesting support (not request).

Id	11	
Template	Free Form	
Fields	FF	The sensor shall change states from NOMINAL to TRANSITION when the autopilot is not requesting support (not request).

Id	11.1	
Template	Sensor State Change	
Fields	PS	0
	NS	1
	RE	false
	LI	false
	MO	mode

Requirement 12

The sensor shall change states from FAULT to TRANSITION when the autopilot is not requesting support (not request) and limits are not exceeded (not limits).

Id	12	
Template	Free Form	
Fields	FF	The sensor shall change states from FAULT to TRANSITION when the autopilot is not requesting support (not request) and limits are not exceeded (not limits).

Id	12.1	
Template	Sensor State Change	
Fields	PS	2
	NS	1
	RE	false
	LI	false
	MO	mode

Requirement 13

The sensor shall change states from TRANSITION to NOMINAL when the autopilot is requesting support (request) and the autopilot reports the correct active mode (mode).

Id	13	
Template	Free Form	
Fields	FF	The sensor shall change states from TRANSITION to NOMINAL when the autopilot is requesting support (request) and the autopilot reports the correct active mode (mode).

Id	13.1	
Template	Sensor State Change	
Fields	PS	1
	NS	0
	RE	true
	LI	limits
	MO	true

C.3 Challenge 2

Requirement 1

When Reset is True and the Initial Condition (ic) is bounded by the provided Top and Bottom Limits, the Output (yout) shall equal the Initial Condition (ic).

Id	1	
Template	Free Form	
Fields	FF	When Reset is True and the Initial Condition (ic) is bounded by the provided Top and Bottom Limits, the Output (yout) shall equal the Initial Condition (ic).

Id	1.1	
Template	Conditional Event Logic	
Fields	TR	true
	IC	reset==true && ic<=TL && ic>=BL
	PC	yout==ic
	TI	1 frames
Answers	Q1	No
	Q2	Yes

Requirement 2

The Output (yout) shall be bounded by the provided Top and Bottom limits.

Id	1	
Template	Free Form	
Fields	FF	The Output (yout) shall be bounded by the provided Top and Bottom limits.

Id	1.1	
Template	Conditional Event Logic	
Fields	TR	true
	IC	true
	PC	yout<=TL && yout>=BL
	TI	1 frames
Answers	Q1	No
	Q2	Yes

Requirement 3

When in normal operation, the output shall be the result of the equation, $yout = T/2(xin + xinpv) + ypv$.

Id	1	
Template	Free Form	
Fields	FF	When in normal operation, the output shall be the result of the equation, yout = T/2 (xin + xinpv) + ypv.

Id	1.1	
Template	Conditional Event Logic	
Fields	TR	true
	IC	reset==false && previous(yout)<=TL && previous(yout)>=BL
	PC	yout==T/2 (xin + xinpv) + ypv
	TI	1 frames
Answers	Q1	No
	Q2	Yes

Requirement 4

The Output of this function shall approximate the integration of the value of the input signal over time within a specified tolerance.

Id	1	
Template	Free Form	
Fields	FF	The Output of this function shall approximate the integration of the value of the input signal over time within a specified tolerance.

Id	1.1	
Template	Free Form	
Fields	FF	After 10 seconds of Computation at an execution frequency of 10 hz, the Output should equal 10 within a +/- 0.1 tolerance, for a Constant Input (xin=1.0), and the sample delta time T=0.1 seconds when in normal mode of operation.

Id	1.1.1	
Template	Conditional Event Logic	
Fields	TR	time==100
	IC	reset==false && previous(yout)<=TL && previous(yout)>=BL && xin==1.0 && T==0.1
	PC	—yout=-10—¡=0.1
	TI	1 frames
Answers	Q1	No
	Q2	Yes

Id	1.2	
Template	Free Form	
Fields	FF	Over a 10 second computational duration at an execution frequency of 10 hz, the Output should equal the sine of time t, sin(t), where time is defined as a vector from 0 to 10 by increments of 0.1 seconds within a +/- 0.1 tolerance for an input equal to the cosine of time t, cos(t), with the sample delta time T=0.1 seconds when in normal mode of operation.

Id	1.2.1	
Template	Conditional Event Logic	
Fields	TR	true
	IC	time¡100 && reset==false && previous(yout)<=TL && previous(yout)>=BL && xin==cos(time) && T==0.1
	PC	—yout-sin(time)—¡=0.1
	TI	1 frames
Answers	Q1	No
	Q2	Yes

C.4 Challenge 3

Requirement 1

The Inner Loop Roll Regulator Shall not command angular roll accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1	
Template	Free Form	
Fields	FF	The Inner Loop Roll Regulator Shall not command angular roll accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1.1	
Template	Maximum Acceleration	
Fields	R	lcvdt_cmd_fcs_dps2
	SC	50
	U	deg/sec^2

Requirement 2

The Inner Loop Pitch Regulator Shall not command angular pitch accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1	
Template	Free Form	
Fields	FF	The Inner Loop Pitch Regulator Shall not command angular pitch accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1.1	
Template	Maximum Acceleration	
Fields	R	mcvdt_cmd_fcs_dps2
	SC	50
	U	deg/sec^2

Requirement 3

The Inner Loop Yaw Regulator Shall not command angular yaw accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1	
Template	Free Form	
Fields	FF	The Inner Loop Yaw Regulator Shall not command angular yaw accelerations greater than the capability of the system (50 deg/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1.1	
Template	Maximum Acceleration	
Fields	R	ncvdt_cmd_fcs_dps2
	SC	50
	U	deg/sec^2

Requirement 4

The Inner Loop Airspeed Regulator Shall not command translational axial accelerations greater than the capability of the system (32 ft/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1	
Template	Free Form	
Fields	FF	The Inner Loop Airspeed Regulator Shall not command translational axial accelerations greater than the capability of the system (32 ft/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1.1	
Template	Maximum Acceleration	
Fields	R	xcvdt_cmd_fcs_dps2
	SC	32
	U	ft/sec^2

Requirement 5

The Inner Loop Height Regulator Shall not command translational height accelerations greater than the capability of the system (32 ft/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1	
Template	Free Form	
Fields	FF	The Inner Loop Height Regulator Shall not command translational height accelerations greater than the capability of the system (32 ft/sec2) for durations exceeding 100 frames (1 second @ 100 hz).

Id	1.1	
Template	Maximum Acceleration	
Fields	R	hcvdt_cmd_fcs_dps2
	SC	32
	U	ft/sec^2

Requirement 6

The Inner Loop Roll Regulator Shall not command transient changes in angular roll acceleration greater than 50 deg/sec2/sec.

Id	6	
Template	Free Form	
Fields	FF	The Inner Loop Roll Regulator Shall not command transient changes in angular roll acceleration greater than 50 deg/sec2/sec.

Id	6.1	
Template	Transient Change	
Fields	R	lcvdt_cmd_fcs_dps2
	SC	50
	U	deg/sec^2/sec

Requirement 7

The Inner Loop Pitch Regulator Shall not command transient changes in angular pitch acceleration greater than 50 deg/sec2/sec.

Id	7	
Template	Free Form	
Fields	FF	The Inner Loop Pitch Regulator Shall not command transient changes in angular pitch acceleration greater than 50 deg/sec2/sec.

Id	7.1	
Template	Transient Change	
Fields	R	mcvdt_cmd_fcs_dps2
	SC	50
	U	deg/sec^2/sec

Requirement 8

The Inner Loop Yaw Regulator Shall not command transient changes in angular yaw acceleration greater than 50 deg/sec2/sec.

Id	8	
Template	Free Form	
Fields	FF	The Inner Loop Yaw Regulator Shall not command transient changes in angular yaw acceleration greater than 50 deg/sec2/sec.

Id	8.1	
Template	Transient Change	
Fields	R	ncvdt_cmd_fcs_dps2
	SC	50
	U	deg/sec^2/sec

Requirement 9

The Inner Loop Airspeed Regulator Shall not command transient changes in translational axial acceleration greater than 32 ft/sec2/sec.

Id	9	
Template	Free Form	
Fields	FF	The Inner Loop Airspeed Regulator Shall not command transient changes in trans- lational axial acceleration greater than 32 ft/sec2/sec.

Id	9.1	
Template	Transient Change	
Fields	R	xcvdt_cmd_fcs_dps2
	SC	32
	U	ft/sec^2/sec

Requirement 10

The Inner Loop Roll Regulator Shall not command transient changes in angular roll acceleration greater than 50 deg/sec2/sec.

Id	10	
Template	Free Form	
Fields	FF	The Inner Loop Roll Regulator Shall not command transient changes in angular roll acceleration greater than 50 deg/sec2/sec.

Id	10.1	
Template	Transient Change	
Fields	R	hcvdt_cmd_fcs_dps2
	SC	32
	U	ft/sec^2/sec

C.5 Challenge 5

Requirement 1

The maximum value of the NN output, z, shall always be less than or equal to 1.1, regardless of the input values.

Id	1	
Template	Free Form	
Fields	FF	The maximum value of the NN output, z, shall always be less than or equal to 1.1, regardless of the input values.

Id	1.1	
Template	Output Value	
Fields	RO	<=
	SC	1.1

Requirement 2

The minimum value of the NN output, z, shall always be greater than or equal to -0.2, regardless of the input values.

Id	2	
Template	Free Form	
Fields	FF	The maximum value of the NN output, z, shall always be greater than or equal to -0.2, regardless of the input values.

Id	2.1	
Template	Output Value	
Fields	RO	>=
	SC	-0.2

Requirement 3

Using a first order finite backward difference equation, the spatial derivatives of $\Delta z/\Delta xt = (z(n,1) - z(n-1))/(xt(n,1) - xt(n-1,1)$ and $\Delta z/\Delta yt = (z(n,1) - z(n-1))/(yt(n,1) - yt(n-1,1)$ shall never exceed a top bound of +10 or bottom bound of -35 (e.g. $-35 <= \Delta z/\Delta(xt, yt) <= 10$), where n denotes an index to the current values and $n-1$ denotes the prior values in the included truth data for xt and yt.

Id	1	
Template	Free Form	
Fields	FF	Using a first order finite backward difference equation, the spatial derivatives of $\Delta z/\Delta xt = (z(n,1) - z(n-1))/(xt(n,1) - xt(n-1,1)$ and $\Delta z/\Delta yt = (z(n,1) - z(n-1))/(yt(n,1) - yt(n-1,1)$ shall never exceed a top bound of +10 or bottom bound of -35 (e.g. $-35 <= \Delta z/\Delta(xt, yt) <= 10$), where n denotes an index to the current values and n-1 denotes the prior values in the included truth data for xt and yt.

Requirement 4

The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, for the equivalent input of (xt, yt).

Id	1	
Template	Free Form	
Fields	FF	The absolute error between the zt truth data and the output z shall never exceed a tolerance of 0.01, for the equivalent input of (xt, yt).

Bibliography

[1] Robust software engineering. https://ti.arc.nasa.gov/tech/rse/. Accessed: 2017-06-22.

[2] Cocosim - automated analysis and compilation framework for simulink/stateflow. https://coco-team.github.io/cocosim/. Accessed: 2017-04-23.

[3] David L Parnas, A John Van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, 1990.

[4] Nancy G Leveson. Software safety: Why, what, and how. *ACM Computing Surveys (CSUR)*, 18(2):125–163, 1986.

[5] William Young and Nancy G Leveson. An integrated approach to safety and security based on systems theory. *Communications of the ACM*, 57(2):31–35, 2014.

[6] Nancy G Leveson. Rasmussen's legacy: A paradigm change in engineering for safety. *Applied ergonomics*, 59:581–591, 2017.

[7] Margaret V. Stringfellow, Brandon D. Owens, Nicolas Dulac, and Nancy G. Leveson. 5.3.1 a safety-driven systems engineering process. *INCOSE International Symposium*, 18(1):605–619, 2008. ISSN 2334-5837. doi: 10.1002/j.2334-5837.2008.tb00830.x. URL http://dx.doi.org/10.1002/j.2334-5837.2008.tb00830.x.

[8] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[9] Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[10] Pericles Loucopoulos and Vassilios Karakostas. *System requirements engineering*. McGraw-Hill, Inc., 1995.

[11] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

[12] Benjamin L Kovitz. *Practical software requirements: a manual of content and style.* Manning Publications Co., 1998.

[13] Betty HC Cheng and Joanne M Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[14] Vincenzo Gervasi and Didar Zowghi. On the role of ambiguity in re. In *REFSQ*, pages 248–254. Springer, 2010.

[15] Fabian de Bruijn and Hans Dekkers. Ambiguity in natural language software requirements: A case study. *Requirements Engineering: Foundation for Software Quality*, pages 233–247, 2010.

[16] Didar Zowghi and Chad Coulin. Requirements elicitation: A survey of techniques, approaches, and tools. In *Engineering and managing software requirements*, pages 19–46. Springer, 2005.

[17] Daniel M Berry and Erik Kamsties. Ambiguity in requirements specification. In *Perspectives on software requirements*, pages 7–44. Springer, 2004.

[18] Luisa Mich, Mariangela Franch, and Pierluigi Novi Inverardi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(2):151–151, 2004.

[19] DC Gause. User driven designthe luxury that has become a necessity. In *A Workshop in Full Life-Cycle Requirements Management. ICRE*, 2000.

[20] Erik Kamsties and B Peach. Taming ambiguity in natural language requirements. In *Proceedings of the Thirteenth International Conference on Software and Systems Engineering and Applications*, 2000.

[21] Erik Kamsties. 11 understanding ambiguity in requirements engineering. 2005.

[22] Adriaan de Groot and Jozef Hooman. Formal requirements engineering. In *Proceedings of the 2nd workshop on Embedded Systems, PROGRESS*, pages 55–59, 2001.

[23] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. Informal and formal requirements specification languages: bridging the gap. *IEEE transactions on Software Engineering*, 17(5):454–466, 1991.

[24] Janis Bubenko, Colette Rolland, Pericles Loucopoulos, and Valeria DeAntonellis. Facilitating" fuzzy to formal" requirements modelling. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 154–157. IEEE, 1994.

[25] Nancy G Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE transactions on software engineering*, 20(9):684–707, 1994.

[26] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE, 1999.

[27] Rachel L Smith, George S Avrunin, Lori A Clarke, and Leon J Osterweil. Propel: an approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 11–21. ACM, 2002.

[28] Rachel L Cobleigh, George S Avrunin, and Lori A Clarke. User guidance for creating precise and accessible property specifications. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 208–218. ACM, 2006.

[29] Cesare Tinelli. Smt-based model checking. In *NASA Formal Methods*, page 1, 2012.

[30] Simulink. https://www.mathworks.com/help/simulink/. Accessed: 2017-07-21.

[31] Stateflow. https://www.mathworks.com/help/stateflow/. Accessed: 2017-07-21.

[32] Zustre. https://github.com/coco-team/zustre. Accessed: 2017-04-23.

[33] Andrew Gacek. Jkind-a java implementation of the kind model checker. *Retrieved June*, 15:2016, 2015.

[34] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[35] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[36] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9): 1305–1320, 1991.

[37] Daniel Pilaud, N Halbwachs, and JA Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*, volume 178, page 188, 1987.

[38] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[39] H Norton Riley. The von neumann architecture of computer systems. *Computer Science Department, California State Polytechnic University*, 1987.

[40] Wesley M Johnston, J.R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, 2004.

[41] Lockheed martin. `http://www.lockheedmartin.com/us.html`. Accessed: 2017-06-22.

[42] Fastparse 0.4.3. `http://www.lihaoyi.com/fastparse/`. Accessed: 2017-07-20.

[43] Ken Bordignon and John Bessolo. Control allocation for the x-35b. In *2002 Biennial International Powered Lift Conference and Exhibit*, 2002.