The CLEAR Way To Transparent Formal Methods

Devesh Bhatt, Anitha Murugesan, Brendan Hall, Hao Ren
Honeywell Advanced Technology, Plymouth, Minnesota, USA
{devesh.bhatt, anitha.murugesan, brendan.hall, hao.ren2}@honeywell.com

Yogananda Jeppu
Honeywell Technology Solutions, Hyderabad, India
Yogananda.Jeppu@honeywell.com

Although Formal Method (FM) based techniques and tools have impressively improved in recent years, the need to train engineers to be accomplished users of formal notation and tailor the tools/workflow to meet objectives of the specific application domain, involves a high initial investment and difficulty to actualize in large, legacy organizations. In this paper, we present our approach to address these challenges in Honeywell Aerospace and share our experiences enroute. Starting with a constrained, structured natural language notation for author requirements and orchestrated with a set of novel in-house tool suite our approach automatically transforms those requirements into a consolidated formal representation to perform rigorous analyses and test generation. While the notation's natural-language flavour provides a 'softer' front end, the tool-suite allows 'transparent' use of formal tools at the back-end without engineers having to know the underlying mathematics and theories. The initial application of our approach across various avionic software systems in Honeywell is well-accepted due to it minimal impact on the existing workflow while leveraging the benefits of formal methods.

1 Introduction

Over the past two decades, formal method techniques have widespread acceptance in many industrial sectors, such as the system and software arenas particularly in the aerospace domain. But, the application of formal methods within large scale industrial projects is still in its infancy. A primary obstacle to the industrial use of formal methods [8] is the perceived mathematics required in the application of these methods. In addition to learning the notation, the complexity involved in easily expressing and validating the 'intents' in the mind of the engineers – such as expressing precedence, chronology, and persistence behaviors – is high when using formal notation. Another obstacle is the level of effort required to integrate tools for formal methods into the existing workflow. Orchestrating formal tools to specific application development workflows, even within the same domain, requires in-depth knowledge of the notations used and their underlying theories; All of these require a steep learning curve and a huge initial investment that naturally discourages large industries from adopting formal methods directly in their development.

In this paper, we present our approach to a 'transparent' formal-methods workflow at Honeywell Aerospace where the initial results are promising. Drawing heavily from our experience with several large-scale avionic systems as well as industrial techniques [7] such as EARS [11], PBR [12], etc. [14, 15, 10, 1, 2], we define a constrained, structured natural language notation to capture requirements, called CLEAR (Constrained Language Enhanced Approach to Requirements). Further, our tool *Text2Test*, developed in-house, is specifically purposed to automatically convert the CLEAR requirements into an unambiguous, intermediate, consolidated formal representation; this can be handed down

Submitted to:
SOS 2007

as-is to existing Honeywell tool suites, as well as be easily transformed for other tools suites to perform formal back-end analysis such as model-checking, consistency and completeness analysis, and automated test generation.

We have been reasonably successful in our initial venture of introducing this notation-tool suite combination in several avionic software systems in Honeywell. We received promising results and positive feedback since (a) the natural-language flavor of CLEAR provides a 'soft' front-end for engineers to author requirements with minimal disruption to their existing workflow and (b) the automated transformation and analysis of requirements do not necessitate the engineers to know the mathematics and theories underlying the formal tools and techniques. In the following sections, we introduce the notation and tool-suite, illustrating with examples from real aerospace systems and discussing challenges and critical issues encountered enroute.

2 Overall Approach

Over the last several years, we have developed a number of powerful tools that perform automated analysis and test generation [5, 4] at Honeywell Aerospace and have extensively used them in several product groups to claim certain DO-178C objectives [9]. However, it was not easy to deploy them in newer applications, leverage their combined benefits or add new tools due to vast difference between their input/output notations and underlying mathematics. It was challenging to provide extensive training to new engineers not only to use the tools and be adept in those notations, but also in precisely orchestrating the tools and translating notations. Moreover, requirements – a crucial prerequisite for most analysis – when captured in formal notations were not easy to understand and validate with high-confidence.

To overcome these issues, we defined CLEAR (Constrained Language Enhanced Approach to Requirements) notation to capture requirements that has an intuitive, natural-language flavor, whilst provides a rigours formal basis. CLEAR leverages several best practices and state-of-the-art requirements specification approaches from academia and industry. One of our major considerations in designing CLEAR is to ensure compatibility with emerging verification technologies and enable a transparent translation to tool inputs from the requirements.

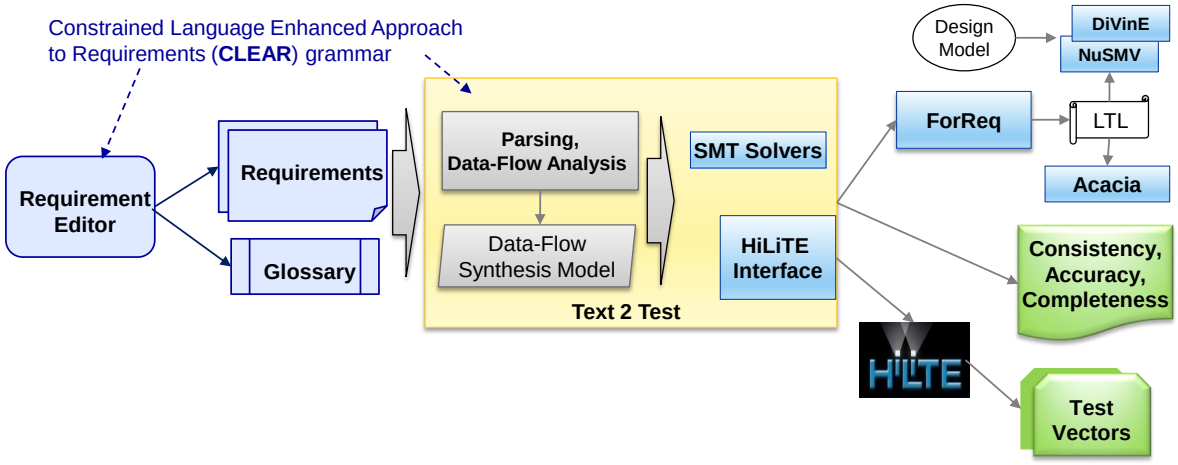

Figure 1: Tool-suite Overview

Figure 1 shows our overall approach and tool-suite that starts with requirements authored in CLEAR. We are currently working on an Eclipse-based comprehensive requirements editor tool that will help authoring requirements in CLEAR notation. The tool provides requirement templates, notation suggestions, and interface to back-end analysis tools, and display results of analysis with just the push of a button.

Once authored in CLEAR, an in-house tool called *Text2Test*, that we specifically developed for this purpose, parses the requirements for syntax and basic semantics. The parsed requirements are then converted into a *data-flow synthesis model* that specifies the input conditions of the requirements as controlling the assignment of expressions to outputs; with timing constructs as part of the data-flow nodes. Then, the tool automatically interfaces with an existing Honeywell's tool – HiLiTE [5, 6] – to provide requirement-based static analysis and test generation. Additionally, Text2Test utilizes public domain SMT Solvers (e.g., Z3) to provide consistency and (limited) completeness analysis on the synthesis model. A variety of arithmetic (including non-linear), logical, and time-based constructs are supported as part of these capabilities to allow the tools to be used for large-scale industrial problems.

Further, an automated translation to Linear Temporal Logic (LTL) can be performed from the data-flow synthesis model using the *ForReq* tool [4, 3]. This tool was developed by Honeywell to provide a transparent interface to several model checkers including DiVinE and NuSMV, and also to tools such as Acacia for realizability analysis.

In summary, the CLEAR notation provides an "on-ramp" for integrating existing and emerging tools and technologies in a way that is transparent to the engineers. Once requirements are authored, there is no intervention required from the engineer during the analysis. Further, the formal representation of the synthesis model allows automated back-end transformation between notations and seamless integration among the tools.

3 CLEAR Notation

Central to our notation is the notion of *requirements as a set of observable properties* of the system under specific conditions. To that end, we defined a set of structures and constructs to specify requirements that have the expressiveness as natural-language as well as the required level of specificity to capture properties of most Aerospace applications. The structures and constructs are influenced by several industry standard requirements notations as well as actual system requirements used in several Honeywell Aerospace applications. This notation, we believe, fully enables the compliance with DO-178C and in particular DO-331 for software high-level requirements.

System Glossary : A *Glossary* of terms such as nouns and verbs used in the requirements have to be defined with basic information such as their name, units, data formats and operational ranges. This not only helps avoid ambiguity and vagueness in requirements, but also helps automate analysis. We have found that glossaries can be often automatically extracted from domain ontologies, dictionaries and interface documents. For instance, several projects had text files, spreadsheets and interface definition documents in which the glossary terms were specified in uniform manner. In other projects, the glossary terms were present in databases with a certain structure. By creating a small file read/write utility script, we were able to easily extract the glossary into a format suitable for CLEAR.

Requirements Structure: Conforming to a simple, consistent structure when writing requirements helps enhance the requirements' quality. To that end, as shown in Table 1, CLEAR provides basic structures (influenced from EARS) and advanced structures (typically used to specify avionic systems), that provide a high-level framework to precisely organize clauses within requirements.

Table 1: Types of Requirement Statements

Type	Usage	Example
Ubiquitous	Unconditional property	*The engine_speed shall be 120 rpm*
Event Driven	System behaviour triggered by events or conditions	*When display_off_button is 'pressed' then display shall be switched off*
State-based	System's behavior in a certain state	*While the microwave_state is 'ON', the interlock shall be true*
Abnormal	Unwanted behavior	*If smoke_level* $>$ *5 then alarm shall be true*
Feature-driven	Feature based system behavior	*Where cruise_control is 'installed', cruise_status shall be 'displayed'*
Complex	Capture events, states and/or features	*While mode is 'On', When gear is 'deployed', then display shall be 'enabled'*
Tabular	Behaviours pertaining to same output(s) in response to mutually exclusive sets of conditions	*System shall compute output based upon TruthTable, input1, input2, output* *row, true, 5, 10* *row, false, 10, 20* ... end
Precedence	Priority based Behaviours (Non-exclusive conditions)	*The value of led shall be per the precedence order* *When alarm is 'enabled', then led shall be 'red'* *When pause is True, then led shall be 'blue' ... end*
Default	Behaviour when conditions of a set of requirements are not satisfied	*When none of the conditions as per {Req1, Req2, Req3} are satisfied, then by default the display shall be 'normal operation'.*
Alternate	Alternate response	*When reservoir_volume is less than 5, then refill_indicator shall be true, otherwise refill_indicator shall be false*
Initial	Initial or startup behaviour	*Initially, the display shall be 'disabled'.*

Notation Constructs: CLEAR notation provides a wide range of constructs to express logical, relational, temporal and arithmetic operations. The constructs can be expressed in both their mathematical form as well as natural-language form such as *is greater than* or $>$, *absolute value of* or $||$, etc. In addition to these common operation constructs, CLEAR also provides advanced constructs typically used in aerospace system requirements such range limiting (*upper/lower limited by*), prioritized signal selection (*select with priority order*), persistence (*persistently be*), state transitions (*transitions from...to*), signal validity (*valid, invalid*), etc. While we have found that these special constructs have a standard interpretation across most Honeywell aerospace applications, we also allow custom definitions depending upon the need of the application domain. These special constructs were defined to allow the requirements author focus on specifying the 'intent', without having to use long, descriptive text to explain it.

Further, CLEAR notation also allows mathematical expressions in both textual and equation form to be captured as/within requirements as recommended in DO-178C. Equations or expressions can be written using valid combinations of constants, variables, and arithmetic operations such as $+, -, *, /$ and $\%$. In addition, to ease the expression of complex mathematical operations the notation provides a number of utility functions such as trigonometric functions, rounding off, etc. For example, Cabin_Altitude shall be $(Cabin\_Altitude/Altitude) * Altitude\_Plane$ is a valid requirement in CLEAR.

4 Verification Tool Chain

Starting with requirements written in CLEAR, various back-end tools for automated requirements analyses, model checking, and test generation can be employed, as suitable to what works to reduce the defects in a particular domain.

Data-Driven Requirement Synthesis : Text2Test parses the requirements authored in CLEAR and creates a data-flow synthesis model. The condition-response structure of each requirement is converted to a *switch* node at the back end – where the condition acts as a control trigger and the choices of responses are selected based on the condition. Though most requirements are typically specified as a response to a condition (*when*...,*then*...), it is however, considered as a 'partial' behavioural description, since there is no specified response for the condition not met. When such a 'partial' specification is encountered, a specialized *switch* node that allows *invalid* value as the missing response is used in the model. Whenever the condition of the switch node is false, the value *invalid* is propagated across the model. Further, nodes for each clause within a requirement for every operation, such as comparisons, arithmetic, etc., are automatically created and connected together to form the data-flow synthesis model. In formal terms, an abstract syntax tree is internally created. Finally, the set of requirement statements that specifies responses of the same output variable are aggregated into a specialized *combiner* node, that ensures if the outputs are consistent and valid values can be propagated as outputs.

Requirements Analysis : Next, Text2Test performs requirement defect analysis over each output variable using an SMT solver and reports the results in an easily readable HTML format. The following defects are identified:

1. *Consistency check* loops through each priority level of an output variable. Let $\{< cond_1^p, resp_1^p > , \ldots, < cond_n^p, resp_n^p >\}$ be the set of condition-response pairs specified for the variable v at the same priority level p. Without loss of generality, we assume all responses are unique, then the consistency check is formulated as

$$\left(\bigoplus(cond_1^p, \ldots, cond_n^p)\right) \vee \left(\neg \bigvee(cond_1^p, \ldots, cond_n^p)\right)$$

   where the first disjunctive clause uses the *xor* operator ($\oplus$) to ensure that one and only one condition holds at a time and the second clause relaxes that logic to allow the case when no condition holds. An SMT formulation of the above is constructed by backward search from the combiner over the model [13].

2. *Domain coverage check* examines the completeness of the input space of an output variable. Let $\{< cond_1, resp_1 >, \ldots, < cond_m, resp_m >\}$ be the set of condition-response pairs for a variable v at all priority levels. The SMT formulation for this analysis is:

$$\bigvee(cond_1, \ldots, cond_m).$$

3. *Incomplete output range check* is performed by comparing the specified output variable range and the range propagated by HiLiTE. Currently this is done for Boolean or Enumeration variables.

4. *Default requirement check* performs a group of correctness checks against the default requirement. Default requirements are checked for redundancies (covered by other requirements), incorrect references to requirements, and duplication (multiple default requirements).

An illustrative example of requirement is given below and its requirements analysis report is shown in Figure 2.

Example: The following requirements are specified on the output variable *LedColor* with input variables *comp_state*, *power_button*, and *power_button*. All the variables in this example are of Enumeration type with enumeration values are specified separately (omitted here).

- ***ExceptionReq:*** If comp_state is 'abnormal', then LedColor shall be 'Red'
- ***StateReq:*** While comp_state is 'idle', LedColor shall be 'Yellow'
- ***NormalReq:*** When power_button is 'ON', then LedColor shall be 'Green'
- ***DefaultReq:*** When none of the conditions as per requirements {ExceptionReq, StateReq, NormalReq} are satisfied, then by default LedColor shall be 'White'

Output Variable : LedColor	
Requirements : ExceptionReq,StateReq,NormalReq,DefaultReq	
Errors in Default Requirements	
DefaultReq	Reference requirements are not specified correctly
Inadquate Domain Coverage	
Set 1	
comp_state	dynamic
power_button	OFF
Set 2	
comp_state	static
power_button	OFF
Set 3	
comp_state	enroute
power_button	OFF
Requirements Inconsistency	
Set: 1	
StateReq	Yellow
NormalReq	Green

Figure 2: Defect report from Text2Test

4.1 Requirement-Based Test Generation

The synthesis model of requirements is then sent to the HiLiTE tool for automated test generation. HiLiTE generates specific tests at the model level for each block embedded in the model, using either heuristic test case templates via backward propagation, or the formal specification of equivalence class via SMT-solving [13]. In the former approach, each equivalence class of a block's behavior (test requirement) is associated with a set of test case templates heuristically selected as a specific combination of values for the input(s) and output(s) of the block under test (BUT). Backward and forward propagation search through the computations of upstream and downstream blocks generates a test vector in terms of

model inputs and outputs to ensure controllability of the BUT inputs and observability of the expected BUT output. In the later approach, SMT-solving embodies formulating test case generation constraints from both equivalence class rules of the BUT and the blocks within its upstream sub-graph, as well as the connectivity, into an SMT problem. Therefore, constraints can be solved together to find a satisfying solution which excludes any conflicts. The formal specification of equivalence classes enables the search on a complete solution domain, empowered by the capability of SMT-solving on scenarios involving mixed data-types, linear/nonlinear computation, and complex model structure.

5 Discussion

We have successfully performed several proof of concepts on industrial case studies within Honeywell Aerospace using CLEAR notation and the associated tool chain. While there has been positive feedback from engineers and willingness to deploy it in more applications, some challenges and issues surfaced. At first, engineers felt that it was an overhead to capture requirements with the level of specificity required by CLEAR since domain experts seems to have the 'tribal knowledge' to interpret requirements without the level of detail. However, in the process of writing in CLEAR notation and the analysis feedback obtained from the tools, they were convinced of why capturing those details will save their time later in the development process. While most requirements were expressible in CLEAR, some applications required newer clauses. However, the extendibility of the notation allowed us to easily add them. Another major concern in large organization like Honeywell is that the developments are often enhancements to legacy applications rather than green-field. When we applied our approach in such applications, the specificity and format of the newly written requirements in CLEAR did not match with the legacy ones. Also, those legacy requirements could not be compositionally analyzed using our tools. We are discussing with such teams to see if they can standardize their requirements in a way that is parsable by the tools. Further, to help engineers with authoring the requirements in CLEAR, we are developing an authoring tool with 'intellisense' that suggests structures, clauses and terms. Hence, at Honeywell Aerospace, CLEAR notation has paved ways for transparently including formal method based analysis and tools with minimal disruption to the existing workflow.

References

[1] (2017 (accessed December 13, 2017)): *D-RisQ software systems*. `http://www.drisq.com/`.

[2] (2017 (accessed December 13, 2017)): *QRA*. `https://qracorp.com/`.

[3] Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran & Tomáš Kratochvíla (2016): *Analysing sanity of requirements for avionics systems*. Formal Aspects of Computing, pp. pp–1.

[4] Jiří Barnat, Jan Beran, Luboš Brim, Tomáš Kratochvíla & Petr Ročkai (2012): *Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs*. In: Formal Methods for Industrial Critical Systems, Springer, pp. 78–92.

[5] D. Bhatt, S. Hickman, K. Schloegel & D. Oglesby (2007): *An Approach and Tool for Test Generation from Model-based Functional Requirements*. In: Proc. 1st International Workshop on Aerospace Software Engineering.

[6] Devesh Bhatt, Gabor Madl, David Oglesby & Kirk Schloegel (2010): *Towards scalable verification of commercial avionics software*. In: Proceedings of the AIAA Infotech@ Aerospace Conference.

[7] Ronald S Carson (2015): *Implementing structured requirements to improve requirements quality*. In: INCOSE International Symposium, 25, Wiley Online Library, pp. 54–67.

[8] Jennifer A Davis, Matthew Clark, Darren Cofer, Aaron Fifarek, Jacob Hinchman, Jonathan Hoffman, Brian Hulbert, Steven P Miller & Lucas Wagner (2013): *Study on the barriers to the industrial adoption of formal methods*. In: *International Workshop on Formal Methods for Industrial Critical Systems*, Springer, pp. 63–77.

[9] RTCA Inc. (2011): *RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification*.

[10] Graham Jolliffe (2010): *Cost-efficient methods and processes for safety relevant embedded systems (CESAR)–an objective overview*. *Making Systems Safer*, pp. 37–50.

[11] Alistair Mavin, Philip Wilkinson, Adrian Harwood & Mark Novak (2009): *Easy approach to requirements syntax (EARS)*. In: *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, IEEE, pp. 317–322.

[12] Patrice Micouin (2008): *Toward a property based requirements theory: System requirements structured as a semilattice*. *Systems engineering* 11(3), pp. 235–245.

[13] Hao Ren, Devesh Bhatt & Jan Hvozdovic (2016): *Improving an Industrial Test Generation Tool Using SMT Solver*. In: *NASA Formal Methods Symposium*, Springer, pp. 100–106.

[14] Natarajan Shankar & Wilfried Steiner (2016): *ARSENAL: Automatic Requirements Specification Extraction from Natural Language*. In: *NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, 9690, Springer, p. 41.

[15] Roopak Sinha, Sandeep Patil, Cheng Pang, Valeriy Vyatkin & Barry Dowdeswell (2015): *Requirements engineering of industrial automation systems: Adapting the CESAR requirements meta model for safety-critical smart grid software*. In: *Industrial Electronics Society, IECON 2015-41st Annual Conference of the IEEE*, IEEE, pp. 002172–002177.