Formal Requirements Elicitation with FRET

Dimitra Giannakopoulou[1], Anastasia Mavridou[2], Thomas Pressburger[1], Julian Rhein[2], Johann Schumann[2], and Nija Shi[2]

[1] NASA Ames Research Center, CA, USA
{dimitra.giannakopoulou,tom.pressburger}@nasa.gov
[2] SGT, NASA Ames Research Center, CA, USA
{anastasia.mavridou, johann.m.schumann, nija.shi}@nasa.gov

Abstract. FRET is a tool for writing, understanding, formalizing and analyzing requirements. Users write requirements in an intuitive, restricted natural language, called FRETISH, with precise, unambiguous meaning. For a FRETISH requirement, FRET: 1) produces natural language and diagrammatic explanations of its exact meaning, 2) formalizes the requirement in logics, and 3) supports interactive simulation of produced logic formulas to ensure that they capture user intentions. FRET connects to analysis tools by facilitating the mapping between requirements and models/code, and by generating verification code. FRET is available open source at https://github.com/NASA-SW-VnV/fret; a video can be accessed at : https://tinyurl.com/fretForREFSQ.

1 Introduction

Requirements engineering is a central step in the development of safety-critical systems. The vision for NASA Ames' Formal Requirements Elicitation Tool (FRET) is 1) to make the writing, understanding, and debugging of formal requirements as natural and intuitive as possible, 2) to seamlessly connect to powerful external tools for analysis, and 3) to support the correction of requirements suggested by analysis results. FRET users have been limited to teams within NASA Ames and external collaborators, but with FRET's recent open sourcing, we hope to obtain feedback and contributions from the wider research community. This paper presents the features of the first FRET release, which form a solid basis for extensions and further development.

In practice, requirements are typically written in natural language, which is ambiguous and consequently not amenable to formal analysis. Since formal, mathematical notations are unintuitive, requirements in FRET are entered in a restricted natural language named FRETISH. FRET helps users write FRETISH requirements both by providing grammar information and examples during editing, but also through English and diagrammatic explanations to clarify subtle semantic issues. For each requirement, FRET automatically produces formulas that can be used by analysis tools at all phases of the software lifecycle. An extensive verification framework ensures that the generated formulas conform to the semantics of the FRETISH language [8]. Moreover, FRET supports the mapping of high-level requirements to the signals or variables that appear in software models or code. FRET then exports verification code that can be used directly by a variety of analysis tools.

Novelty. FRET incorporates ideas from several existing approaches to requirements engineering. The structure of FRETISH requirements includes features from the Specifica-

tion Pattern System (SPS) [4], and the Easy Approach to Requirements Syntax (EARS, [12]), implemented in tools like Prospec [7], SPIDER [10], SpeAR [6], and EARS-CTRL [11]. In the commercial tool STIMULUS [9], requirements are built by dragging and dropping phrases. The ASSERT™ [3] tool uses the constrained natural language SADL for formalizing domain ontologies, and a requirements language SRL that can express conditions, including temporal conditions, on monitored variables, and constraints on controlled variables. The uniqueness of FRET lies in its main goal: to be an open source, extensible requirements platform that can connect to external requirements analysis tools. As such, the FRETISH language and the formalization capabilities aim at being inclusive, and for this reason are modular, and extensible. For example, FRET produces formalizations in both future- and past-time metric temporal logics. Since we plan on using FRET in safety-critical contexts, ensuring correctness of the supported formalizations is key.

2 Interacting with FRET

This section describes a user's end-to-end interaction with FRET through an example from the publicly-available Lockheed Martin Cyber Physical Systems (LMCPS) challenge [5]. The application of FRET to LMCPS is, to date, the largest FRET case study [13]. FRET's entry point is a dashboard that summarizes the status of selected projects, and provides a hierarchical view of all requirements, as shown in Figure 1 for LMCPS. Requirements can also be displayed in standard tabular form.

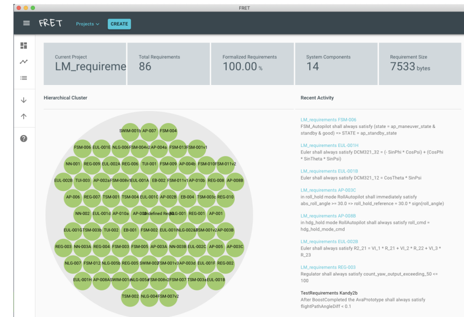

Fig. 1. FRET dashboard

Requirements Elicitation. Figure 2 illustrates FRET's requirements elicitation interface. FRETISH requirement **[AP-002a]**:"in roll_hold mode RollAutopilot shall always satisfy autopilot_engaged & no_other_lateral_mode" expresses the natural language description included in the "Rationale and Comments" field. i.e., that the autopilot should be engaged and no other lateral mode should be active when the Roll Autopilot is in roll hold mode. The interface window consists of the editor, on the left, and a help tab on the right (gray background). The FRETISH grammar, displayed as "railroad diagrams", is accessible from this view by clicking on the question mark.

A FRETISH requirement description is automatically parsed into six sequential fields, with the FRET editor dynamically coloring the text corresponding to the fields as the requirement is typed in (Figure 2): *scope*, *condition*, *component*, *shall*, *timing*, and *response*. Help and examples on each specific field can be displayed in the help tab by clicking on the corresponding field bubble. The mandatory *component* field specifies the component that the requirement applies to (RollAutopilot). The *shall* keyword states that the component behavior must conform to the requirement. The *response* field currently is of the form *satisfy* R, where R is a non-temporal Boolean-valued expression.

Field *scope* (optional) states that the requirement is only relevant in specific scopes of the system behavior, for example when the system is "in roll_hold mode". The Boolean

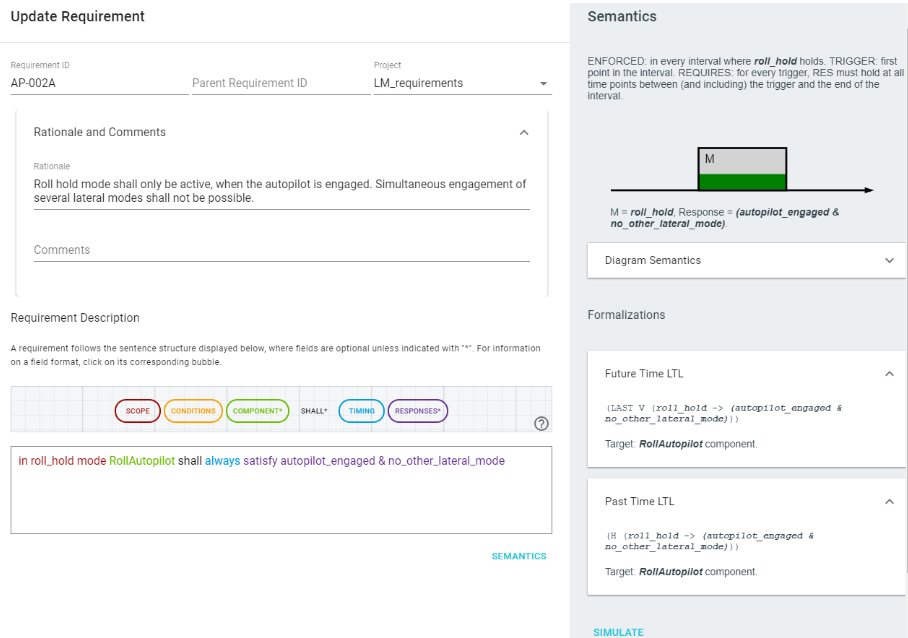

Fig. 2. FRET editor with formalizations and explanations in the help tab.

expression field *condition* (optional) states that, within the specified mode, the requirement becomes relevant only from the point where the condition becomes true. When, as in our example, *condition* is omitted, there is no such restriction. Field *timing* (optional) specifies at which points the response must occur, for example "always", meaning at all points where the system is "in roll_hold mode". Default timing is *eventually*.

By clicking SEMANTICS, the help tab displays various explanations of the requirement, as well as temporal formulas. The diagram of Figure 2 illustrates that the requirement is only relevant within the grayed box M (M represents intervals where the Autopilot is in "roll_hold" mode). The green band states that "autopilot_engaged & no_other_lateral_mode" is required to hold at all time points within the gray box.

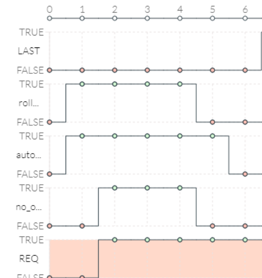

Fig. 3. Screenshot of requirements visualization for AP-002a (Figure 2)

Requirements Visualization. Getting a requirement with temporal relationships right is a tricky and subtle task. Errors and misunderstandings might creep into the formulation, resulting in a requirement that does not correctly reflect the temporal interdependencies of the involved signals. Based upon the graphical signal representation commonly used in digital electronics, we developed an interactive requirements visualizer in FRET, available by clicking SIMULATE in the semantics view of the help tab (see Figure 2). Given a FRET requirement, it shows temporal traces of each of the signals (variables) involved as well as the valuation of the requirement for each point in time (see Figure 3). The user can interactively modify the input signals;

the valuation of the requirement is updated automatically and thus makes it possible for the user to visually inspect the temporal behavior of the requirement. The valuation is computed based on the construction of a finite state machine from the input trace, which is then verified against the LTL formalization of the requirement, using NuSMV [1].

Requirements Analysis. FRET's main purpose is to facilitate the elicitation of unambiguous requirements. For analysis, it allows users to export requirements in formats that can be digested by external analysis tools. FRET currently connects to the CoCoSim tool [2] for analysis of Simulink models, and through CoCoSim to the Kind2, Zustre, and Simulink Design Verifier (SLDV) tools for analysis.

To analyze requirements against an implementation as model or code, one needs to associate the requirement variables, which are at a high level, with variables in the model or code (signals in the case of Simulink). Moreover, FRET needs to generate verification code that can be understood by the target analysis tool. To connect with CoCoSim, FRET transforms requirements into CoCoSpec code. In this process, FRET supports importing Simulink model information provided by Cocosim, and association of high-level requirements with target model signals and components (see Figure 4).

FRET Component: Autopilot — Corresponding Model Component: ap_12BAdapted/GlobalScope

FRET Variable Name	Model Variable Name	Variable Type*	Data Type*	Description
ALTITUTE		Internal	boolean	
AUTOPILOT_ENGAGED	APeng	Input		
EPSILON		Internal	double	
HDG		Mode	boolean	
HDG_STEADY_STATE		Internal	double	

Rows per page: 11-15 of 31

Fig. 4. Associating FRETISH requirements with Simulink models.

3 FRET architecture

FRET is implemented mainly in JavaScript as an Electron JS app. Electron JS[3] is a framework for creating desktop-suite applications by using web development programming languages. Electron JS uses two main technologies: the Node.js runtime and the Chromium web browser. Its file system provided through the Node.js API is compatible with Linux, Mac OS, and Windows. FRET's interactive interface was developed with the React JavaScript library[4]. FRET uses PouchDB[5]as an in-browser database that also runs in Node.js. FRET's architecture is illustrated in Figure 5. This section reviews the main modules in the architecture.

Offline Formalization. This component of the FRET architecture is described in detail in [8]. Formalization of FRETISH requirements is performed by the FORMALIZER

[3] https://electronjs.org/ [4] https://reactjs.org/ [5] https://pouchdb.com/learn.html

component. Formalization is performed based on semantic template keys, which are valuations of the fields that make up each FRETISH requirement. For example, the template key for requirement **[AP-002a]** is [*in*, *null*, *always*], meaning that the scope is "in mode", condition is omitted, and timing is "always". For each template key, the FORMALIZER generates a variety of mathematical formulas, as well as English language explanations and diagrams, which are all saved in a cache. Note that all these artefacts are templates that contain variables. These variables get instantiated by FRET to capture the details of specific requirements, as will be described later.

FORMALIZATION VERIFIER is a modular, extensible framework, which provides assurance that formulas generated by the FORMALIZER capture the intended semantics. It implements 1) a module that generates traces, i.e., example executions over which to interpret formulas; 2) a module that, given a trace and a template key, generates an expected value of true or false based on the semantics of the FRETISH language, 3) a module which interprets formulas over traces and compares the outcome to expected values and 4) a module that compares future and past-time formulas generated for the same template key, for equivalence.

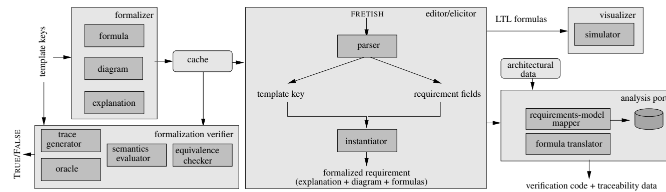

Fig. 5. Architecture of FRET

Requirements Elicitation. The bulk of the work in formalizing FRETISH requirements is performed offline and cached. When users write a requirement, the PARSER identifies the corresponding template key, and the values of the requirement fields, which instantiate the template key. The INSTANTIATOR uses the template key to fetch correct artefacts from the cache, and uses the requirement fields to appropriately instantiate them. The produced LTL formulas can be explored interactively through the SIMULATOR.

Analysis Portal. This component connects FRET with analysis tools. The user needs to define mappings between FRETISH variables and model variables, as well as additional information such as variable types, which is not relevant at a high level. Module REQUIREMENTS-MODEL MAPPER supports this process and stores the provided information in a database. It is also able to import architectural data about model components and signals, when available, to facilitate this task. Module FORMULA TRANSLATOR currently uses information from the mapper, and the past-time LTL formalization of a requirement, to generate Cocospec code, as well as traceability data. The latter is used to report analysis results in the context of FRETISH requirements. This component could also be used in the context of other tools, and is described in detail in [14].

4 Applications and Future Work

As mentioned, the LMCPS case study is the largest FRET case study to date. FRET is currently being used by a mission within the NASA Ames Research Center. We are

working closely with mission developers to help them with but also monitor their use of the tool. We have noticed, for example, that developers initially need help from us to capture requirements in FRETISH and understand the semantic nuances of the fields that are supported. However, their requirements fall into recurring patterns, so they become effective with the use of FRETISH quite fast. For this reason, we are considering ways of supporting new FRETISH users, by, for example, displaying typical requirement patterns within a domain or project, and allowing users to import patterns within the editor and customize as needed. More generally, now that FRET has its basic features established, we are focusing on improving the interaction with users both in editing and correcting requirements. Similarly, we are working on additional analysis tools/algorithms to integrate with FRET; for example, we have been working on providing support for checking requirements realizability.

References

1. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer **2**(4), 410–425 (Mar 2000).
2. CoCo-team: CoCoSim – automated analysis framework for Simulink. Customized, closed version based on the open source version. https://github.com/coco-team/cocoSim2
3. Crapo, A., Moitra, A., McMillan, C., Russell, D.: Requirements capture and analysis in ASSERT(TM). In: Proc. RE. pp. 283–291, IEEE (2017).
4. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. 21st ICSE. pp. 411–420. ACM (1999).
5. Elliott, C.: On example models and challenges ahead for the evaluation of complex cyber-physical systems with state of the art formal methods V&V, Lockheed Martin Skunk Works. In: Proc. S5 (2015), http://mys5.org/Proceedings/2015/Day_1/2015-S5-Day1_1405_Elliott.pdf
6. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: SpeAR V2.0: Formalized past LTL specification and analysis of requirements. In: Proc. NFM 2017, 2017, pp. 420–426 (2017).
7. Gallegos, I., Ochoa, O., Gates, A., Roach, S., Salamah, S., Vela, C.: A property specification tool for generating formal specifications: Prospec 2.0. In: Proc. SEKE. pp. 273–278 (2008)
8. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Generation of formal requirements from structured natural language. In: REFSQ2020 (2020)
9. Jeannet, B., Gaucher, F.: Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In: Proc. ERTS (2016).
10. Konrad, S., Cheng, B.H.C.: Facilitating the construction of specification pattern-based properties. In: Proc. RE, pp. 329–338. IEEE (2005).
11. Lúcio, L., Rahman, S., Cheng, C.H., Mavin, A.: Just formal enough? Automated analysis of ears requirements. In: Proc. NFM. pp. 427–434. Springer (2017)
12. Mavin, A.: Listen, then use EARS. IEEE Software **29**(2), 17–18 (Mar 2012).
13. Mavridou, A., Bourbouh, H., Garoche, P.L., Hejase, M.: Evaluation of the FRET and CoCoSim tools on the ten Lockheed Martin Cyber-Physical Challenge Problems. Tech. Rep. TM-2019-220374, NASA (2019)
14. Mavridou, A., Bourbouh, H., Garoche, P.L., Giannakopoulou, D., Pressburger, T., Schumann, J.: Bridging the gap between requirements and simulink model analysis. Under submission, Posters and Tools Track, REFSQ2020 (2020)