Simplifying Requirements Formalization for Resource-Constrained Mission-Critical Software

Carlos Mão de Ferro
LASIGE, Faculdade de Ciências, Universidade de Lisboa
carlos@maodeferro.pt

Anastasia Mavridou
KBR Inc.
NASA Ames Research Center
anastasia.mavridou@nasa.gov

Michael Dille
KBR Inc.
NASA Ames Research Center
michael.dille@nasa.gov

Francisco Martins
Universidade dos Açores
francisco.cc.martins@uac.pt

***Abstract*—Developing critical software requires adherence to rigorous software development practices, such as formal requirement specification and verification. Despite their importance, such practices are often considered as complex and challenging tasks that require a strong formal methods background. In this paper, we present our work on simplifying the formal requirements specification experience for resource-constrained mission critical software through the use of structured natural language. To this end, we connect NASA's FRET, a formal requirement elicitation and authoring tool with the Shelley model checking framework for MicroPython code. We report our experience on using these tools to specify requirements and analyze code from the NASA Ames PHALANX exploration concept.**

***Index Terms*—requirements, verification, mission-critical code**

I. Introduction

For many decades, since the very introduction of software-controlled complex systems, formal design and implementation methods have been widely and successfully applied in a broad range of safety-critical and high-reliability applications, from medicine to industrial control to aerospace [1], [2]. Also known as cyber-physical systems (CPS), these applications integrate computation with physical processes, presenting many well-studied challenges [3]. Formalisms such as rigorous software development methodologies, static code analysis, dynamic simulation, and logic-based verification have gained near-universal adoption in large or mission-critical software projects. All of these introduce considerable burden in software development, often accounting for a large portion of project cost and schedule [4]. Aerospace applications in particular, presenting extreme risks to human safety and costly hardware, are among the most ardent adopters, with NASA famously coining the phrase "failure is not an option" [5].

This philosophy, though highly successful at producing reliable systems, has not been without criticism, with the pace and cost of many large development efforts compared harshly to silicon valley's contemporary "agile" and "fail fast" mentality. This has contributed to a recent shift within NASA towards pursuing more commercial and experimental work with shorter timelines and reduced mission-assurance scrutiny. A significant recent example is the Commercial Lunar Payload Services (CLPS) initiative [6], which has contracted commercial sources for instruments, robots, and spacecraft for the surface of the moon. In contrast to traditional missions, these have minimal oversight from NASA, instead being fully built and operated by their providers, many of whom are young entrants lacking the extensive experience and resources to apply complex and often-costly formal verification tools. Despite this, these missions are still held to high expectations of quality assurance and success.

Further complicating matters is a broadly growing trend towards ever more miniaturized and distributed networked systems, two examples of which are multi-robot systems and "internet of things" ubiquitous sensor networks. Beyond additional cost and complexity pressures, these more typically rely on smaller "embedded" processors that lack established toolchains and the computational headroom to apply many popular verification techniques, motivating new approaches. As one such illustrative example, we introduce a NASA planetary surface mission concept called PHALANX [7] which proposes multiple small expendable hardware nodes deployed by a robotic rover or lander to form sensor and communication relay networks, thereby expanding the reach and awareness of the parent vehicle. These nodes must be inexpensive and rapidly developed for different specific missions, yet may still provide critical backbone functionality for key mission goals.

One approach to address this void in embedded systems is to use higher level languages with accompanying development and testing approaches, while avoiding many typical pitfalls of lower level languages. For instance, MicroPython [8] is an implementation of the Python programming language designed for microcontrollers, providing a large subset of standard Python features in a reduced memory footprint. For demonstrative evaluation, we have ported PHALANX to it.

Meanwhile, tools for elicitation and formalism of requirements can simplify and expedite verification. One such tool is the open-source Formal Requirements Elicitation Tool (FRET) [9] which captures requirements using natural language, explains these diagrammatically, automatically generates corresponding linear temporal logic (LTL) formulae for verification, and provides interactive simulation. Another such tool is Shelley [10], a model-checking framework applying LTL on finite traces [11], [12] to validate formalizations

This work was supported by FCT through scholarships SFRH/BD/131418/2017 and SFRH/BI/153747/2019, and the LASIGE Research Unit ref. UIDB/00408/2020 and ref. UIDP/00408/2020. PHALANX concept development was co-led by Uland Wong.

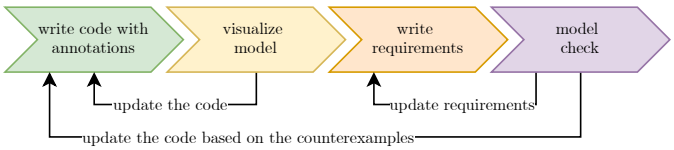

Fig. 1: Toolchain architecture

against a MicroPython model and provide state-machine visualization to assist in correcting requirements.

In this paper, we evaluate the feasibility and benefits of integrating these two tools to perform requirements elicitation, formalization, and analysis directly on MicroPython code, as illustrated in Figure 1. Once requirements are written in FRET, we export their formalizations for digestion by Shelley, via a dedicated parser that matches each requirement label to the corresponding identifier on a MicroPython class. Our contribution therefore is an end-to-end simplified process that supports the specification, understanding and formalization of requirements in the challenging context of resource-constrained mission-critical systems.

II. Modeling in Shelley

Shelley is a model checking framework and offers a simple, yet expressive modeling language that can be used to verify the order of function calls in a programming language. In this paper, we use Shelley to model and verify code written in MicroPython. The process of extracting a Shelley model from a MicroPython class is out of the scope of this paper, and we instead briefly provide an intuition for the process. For each MicroPython class, we use annotations so that developers can describe the expected ordering of calling methods. For instance, a method that can be called at the beginning of the object lifecycle should be annotated with `@op_initial`. In addition, for each method under analysis we use the `return` statement to give information about the next methods that can follow. This way, we enforce the behavior of an object and how it can be used in such a way that we can model check temporal requirements. We also capture if method invocations of objects under analysis are defined by inferring the behavior of each method in the class, described as a regular expression. We use the MicroPython `match` statement to check for exhaustive tests on matches that take the result of a method-invocation under analysis. Finally, the Shelley framework includes a visualization tool that automatically generates behavior diagrams based on the code annotations and the methods' bodies.

A. The Coordinator running example

Our running example is based on a simplification of a piece of the PHALANX application. In this scenario, a mobile surface rover deploys different sensor nodes in order to form a Wireless Sensor Network (WSN). When the deployment is finished, the rover can move away to complete other science tasks. After some time, the rover returns to the deployment area to wirelessly retrieve collected sensor data from the network. While moving, the rover searches for new nodes that are advertising. Every time it finds a new sensor, it will change mode to synchronize with that device and exchange some data.

Listing II.1: Class Coordinator. To make the example concise, we abbreviated `sensor` to the single letter `s`.

```
@reqs(["RQ1", "RQ2", "RQ3"])
@sys(["s"])
class Coordinator:
  def __init__(self):
    self.s = Sensor()

  @op_initial
  def search(self, max):
    for _ in range(max):
      match self.s.available():
        case "connected":
          self.s.connected()
          match self.s.tick():
            case "ready":
              self.s.ready()
              return "sync"
            case "done":
              self.s.done()
              return "sleep"
            case "failed":
              self.s.failed()
        case "failed":
          self.s.failed()
    return "sleep"

  @op
  def sync(self):
    data = self.s.flush()
    # ... handle data ...
    match self.s.tick():
      case "ready":
        self.s.ready()
        return "sync"
      case "done":
        self.s.done()
        return "search"
      case "failed":
        self.s.failed()
        return "sleep"

  @op_final
  def sleep(self):
    # suspend execution
```

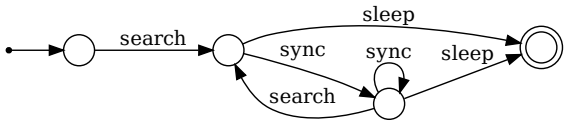

Fig. 2: `Coordinator` diagram based on the class methods and the return statements.

The rover carries a moving node, called the *Coordinator*. Since the Coordinator is unique in the network it might represent a single point of failure. Therefore it is critical that the Coordinator's software is correct.

We now describe the `Coordinator` class as seen in Listing II.1. Each method represents a different state and the return statements correspond to state transitions. For simplicity, we assume only three states (hence only three methods) for the Coordinator class:

- `search`: The rover is searching for new sensor nodes;
- `sync`: The rover established a connection and synchronizes until no more data is available;
- `sleep`: If communication with the sensor fails or if synchronization has completed the rover can save battery by sleeping the Coordinator node.

The Coordinator class issues calls to another object that we model as well. For instance, the `sensor` attribute (abbreviated `s`) corresponds to an instance of `Sensor` (not shown intentionally). The Coordinator periodically polls the `Sensor` device for new commands by issuing the call `sensor.tick()`. Based on the return of this call, a specific state transition will occur also depending on the current state, *i.e.,* which Coordinator's method. Figure 2 is the behavior diagram for the methods of the class `Coordinator` automatically generated by Shelley.

To verify the correct order of objects' calls, Shelley can be used to model check temporal requirements on the automatically extracted model from the `Coordinator` class.

III. Requirements in FRET

In this section, we elaborate on several requirement examples and show how we used FRET to express requirements and clarify subtle semantic issues. As seen in Line 1 of Listing II.1 we use the annotation `@reqs` to list a set of identifiers that correspond to requirements in the FRET framework. A FRETish requirement is described using up to six sequential fields (the * symbol designates mandatory fields): 1) `scope` specifies the time intervals where the requirement is enforced, 2) `condition` is a Boolean expression that triggers the `response` to occur at the time the expression's value becomes true from false, or is true at the beginning of the scope interval, 3) `component*` is the system component that the requirement is levied upon, 4) `shall*` is used to express that the component's behavior must conform to the requirement, 5) `timing` specifies when the response shall happen, subject to the constraints defined in `scope` and `condition` and 6) `response*` is the Boolean expression that the component's behavior must satisfy.

For each requirement example, we show their FRETish version and the corresponding generated formalization. FRET generates Future Time LTL formulas for infinite and finite traces. Shelley reasons only about finite traces.

a) Requirement RQ1: This requirement characterizes the usage of the `sensor` object modeled according to the Sensor class (not shown), which requires issuing `sensor.tick()` before `sensor.ready()`. When using `sensor` in the `Coordinator` class, we want to additionally enforce that we eventually issue `sensor.tick()` after `sensor.ready()` again, meaning that it is not enough to poll the sensor only once.

FRETish:

```
if sensor_ready the COORDINATOR shall eventually satisfy sensor_tick
```

Generated infinite trace LTL formula:

```
((G (((! sensor_ready) & (X sensor_ready)) ->
  (X (F sensor_tick)))) & (sensor_ready -> (F sensor_tick)))
```

Generated finite trace LTL formula:

```
((LAST V (((! sensor_ready) & ((! LAST) & (X sensor_ready))) ->
   (X ((! LAST) U sensor_tick))))
   & (sensor_ready -> ((! LAST) U sensor_tick)))
```

LAST represents the last time point of a bounded execution trace. For each requirement, FRET generates textual and diagrammatic explanations (Figure 3). In the diagram, TC stands for Triggering Condition. The gray rectangle represents the scope interval of the requirement, which is global, i.e., the whole execution trace. The orange rectangle says that the response must hold at least somewhere in this interval.

b) Requirement RQ2: The call `sensor.flush()` is not issued until we have transitioned to the `sync` state first, *i.e.,* by issuing method `sync`.

FRETish:

```
COORDINATOR shall until sync satisfy !sensor_flush
```

Generated infinite trace LTL formula:

```
(sync V ((! sensor_flush) | sync))
```

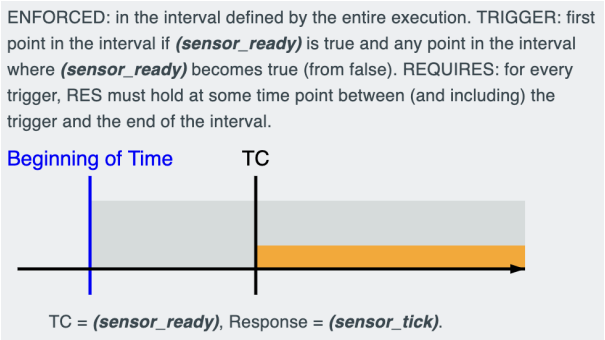

Fig. 3: Semantic explanations for requirement RQ1

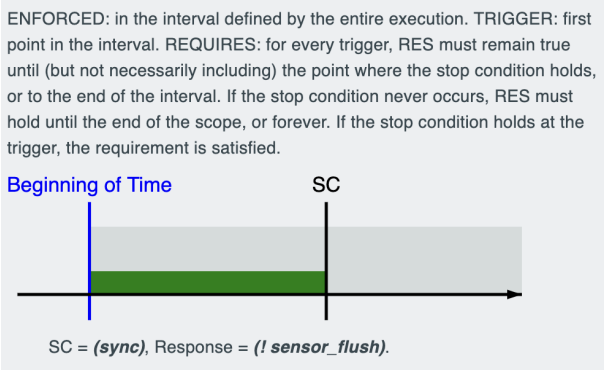

Fig. 4: Semantic explanations for requirement RQ2.

Generated finite trace LTL formula:

```
((sync V ((! sensor_flush) | sync)) | (LAST V (! sensor_flush)))
```

The textual and diagrammatic explanations for requirement RQ2 are shown in Figure 4. SC stands for Stop Condition. The green rectangle says that the response must hold everywhere in this interval. It is interesting to note that originally we attempted to write this requirement directly in LTL without FRET. In this initial formalization attempt we specified a stronger property by using the strong Until operator. Using FRET helped us focus more on the intended semantics of the requirement and formulate it with the Release (V) temporal operator, which does not require the `sync` to happen. As part of the process, we used the FRET simulator to check how different variable valuations affect the valuation of the complete requirement (Figure 5).

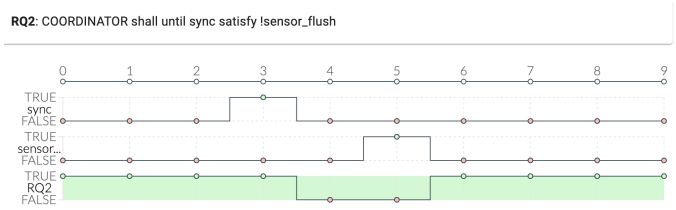

Fig. 5: Simulating requirement RQ2

c) Requirement RQ3: At the next time step after invoking the method `search`, the call `sensor.available()` is issued to check if any sensor node is available to interact (*cf.* Figure 6).

FRETish:

```
Upon search COORDINATOR shall at the next timepoint
satisfy sensor_available
```

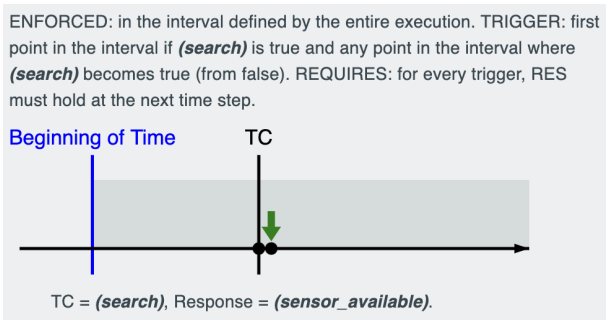

Fig. 6: Semantic explanations for requirement RQ3.

Generated infinite trace LTL Formula:

```
((G (((! search) & (X search)) ->
  (X (X sensor_available)))) & (search -> (X sensor_available)))
```

Generated finite trace LTL Formula:

```
((LAST V (((! search) & ((! LAST) & (X search))) ->
  (X (LAST | (X sensor_available)))))
  & (search -> (LAST | (X sensor_available))))
```

IV. Verification

The last step in our approach is to use the Shelley model checker to verify the temporal formulae against the extracted model. We model checked the `Coordinator` class against the aforementioned three requirements. Shelley only outputs a message when there is a violation, i.e., a requirement is not satisfied by the model. Through the model checking act, we found that there is an error in the `Coordinator` code. Shelley output the following counterexample for requirement RQ3:

```
Error in specification: FAIL TO MEET REQUIREMENT
Formula: (G (((! search) & (X search)) ->
  (X (X sensor.available)))) & (search -> (X sensor.available))
Counter example: search, sensor.available, sensor.connected,
  sensor.tick, sensor.ready, sync, sensor.flush,
  sensor.tick, sensor.done, >search, sleep<
```

From the counterexample, we can observe that it is actually possible to issue `search` immediately followed by `sleep` (which is against the specification). This execution trace is not obvious by just looking at the code. Note that since the `for` loop in Line 9 of Listing II.1 might run zero times, it is possible that the execution jumps straight to Line 24 meaning that the method `sleep` shall be executed immediately after. Shelley can generate behavior diagrams that show the calls inside each method making it easier to understand the code flow. A way to fix this erroneous behavior is to repeat Lines 10 to 23 thus ensuring that the code inside `loop`, which starts with the call `sensor.available()`, runs at least once.

V. Related work

FRET has been used before in various industrial case studies [13]–[15]. However, none of these describe work on specifying requirements for model checking MicroPython mission-critical code. Java PathFinder [16] and the Bandera Tool Set [17], for Java, and JKind [18], for Lustre, are examples of model checkers targeting general-purpose programming languages and focus on concurrency rather than ensuring specific requirements about the behavior of a program. JavaBIP [19] is a framework that uses annotations directly on Java code in order to coordinate existing concurrent software components.

VI. Conclusion and Future Work

We described a process of integrating two tools in order to specify requirements in an intuitive format, generate formal specifications and model check these against MicroPython code. Our use case illustrated how it is possible to apply formal methods to analyze MicroPython code, even for users that do not have a strong background on temporal logics. We believe that our approach will help delivering high-quality software that is correct without the high costs of training a team or using expensive modeling and simulation tools.

In the future, we plan to extend Shelley to support requirements that express bounded time constraints thus leveraging the full power of the FRET framework.

References

[1] S. P. Nanda and E. S. Grant, "A survey of formal specification application to safety critical systems," in *ICICT*, 2019, pp. 296–302.

[2] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "A summary of formal specification and verification of autonomous robotic systems," in *IFM*, ser. LCNS, vol. 11918. Springer, 2019, pp. 538–541.

[3] A. Müller, S. Mitsch, W. Retschitzegger, and W. Schwinger, "Towards CPS verification engineering," in *iiWAS*. ACM, 2020, p. 367–371.

[4] Hillel Wayne. Why don't people use formal methods? [Online]. Available: https://www.hillelwayne.com/post/why-dont-people-use-formal-methods/

[5] G. Kranz, *Failure is not an option : mission control from Mercury to Apollo 13 and beyond* . New York, N.Y: Simon & Schuster, 2000.

[6] NASA. Commercial Lunar Payload Services. [Online]. Available: https://www.nasa.gov/content/commercial-lunar-payload-services

[7] M. Dille, D. Nuch, S. Gupta, S. McCabe, N. Verzic, T. Fong, and U. Wong, "PHALANX: Expendable projectile sensor networks for planetary exploration," in *IEEE Aerospace Conference*, March 2020.

[8] D. George. (2022) MicroPython. [Online]. Available: https://micropython.org

[9] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi, "Formal requirements elicitation with FRET," in *REFSQ-2020*, ser. Tools Track, vol. 2584. CEUR-WS.org, 2020.

[10] C. M. de Ferro, T. Cogumbreiro, and F. Martins, "Shelley, a framework for model checking call ordering on hierarchical systems," to appear in COORDINATION, ser. LNCS. Springer, 2023.

[11] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL semantics for runtime verification," *Journal of Logic and Computation*, vol. 20, no. 3, pp. 651–674, 2010.

[12] G. De Giacomo, R. De Masellis, and M. Montali, "Reasoning on LTL on finite traces: Insensitivity to infiniteness," in *AAAI*. AAAI Press, 2014, p. 1027–1033.

[13] H. Bourbouh, M. Farrell, A. Mavridou, I. Sljivo, G. Brat, L. A. Dennis, and M. Fisher, "Integrating formal verification and assurance: An inspection rover case study," in *NFM*. Springer International Publishing, 2021, pp. 53–71.

[14] M. Farrell, M. Luckcuck, O. Sheridan, and R. Monahan, "Fretting about requirements: Formalised requirements for an aircraft engine controller," in *REFSQ*. Springer International Publishing, 2022, pp. 96–111.

[15] A. Mavridou, H. Bourbouh, D. Giannakopoulou, T. Pressburger, M. Hejase, P.-L. Garoche, and J. Schumann, "The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained," in *RE*, 2020, pp. 300–310.

[16] D. Giannakopoulou and C. S. Pasareanu, "Interface generation and compositional verification in JavaPathfinder," in *FASE*, ser. LNCS, vol. 5503. Springer, 2009, pp. 94–108.

[17] J. Hatcliff and M. B. Dwyer, "Using the Bandera tool set to model-check properties of concurrent Java software," in *CONCUR*, ser. LNCS, vol. 2154. Springer, 2001, pp. 39–58.

[18] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind model checker," in *CAV*. Springer, 2018, pp. 20–27.

[19] S. Bliudze, A. Mavridou, R. Szymanek, and A. Zolotukhina, "Exogenous coordination of concurrent software components with JavaBIP," *Softw. Pract. Exp.*, vol. 47, no. 11, pp. 1801–1836, 2017.