Automated Translation of Natural Language Requirements to Runtime Monitors

Ivan Perez[1] (✉), Anastasia Mavridou[2] (✉), Tom Pressburger[3], Alwyn Goodloe[4], and Dimitra Giannakopoulou[3]*

[1] National Institute of Aerospace, Hampton, Virginia
[2] KBR Inc., NASA Ames Research Center, Moffett Field, California
[3] NASA Ames Research Center, Moffett Field, California
[4] NASA Langley Research Center, Hampton, Virginia
{ivan.perezdominguez,anastasia.mavridou}@nasa.gov

Abstract. Runtime verification (RV) enables monitoring systems at runtime, to detect property violations early and limit their potential consequences. This paper presents an end-to-end framework to capture requirements in structured natural language and generate monitors that capture their semantics faithfully. We leverage NASA's Formal Requirement Elicitation Tool (FRET), and the RV system Copilot. We extend FRET with mechanisms to capture additional information needed to generate monitors, and introduce Ogma, a new tool to bridge the gap between FRET and Copilot. With this framework, users can write requirements in an intuitive format and obtain real-time C monitors suitable for use in embedded systems. Our toolchain is available as open source.

1 Introduction

Safety-critical systems, such as aircraft, automobiles, and power systems, where failure can result in injury or death of a human [23], must undergo extensive assurance. The verification process must ensure that the system satisfies its requirements under realistic operating conditions and that there is no unintended behavior. Verification rests on possessing a precise statement of requirements, arguably one of the most difficult tasks in engineering reliable software.

Runtime verification (RV) [21, 19, 5] has the potential to enable the safe operation of complex safety-critical systems. RV monitors can be used to detect and respond to property violations during missions, as well as to verify implementations and simulations at design time. For monitors to be effective, they must faithfully reflect the mission requirements, which is difficult for non-trivial systems because correctness properties must be expressed in a precise mathematical formalism while requirements are generally written in natural language.

The focus of this paper is to provide an end-to-end framework that takes as input requirements and other necessary data and provides mechanisms to 1) help the user deeply understand the semantics of these requirements, 2) automatically generate formalizations and 3) produce RV monitors that faithfully

* Author contributed to this work prior to joining AWS.

This is a U.S. government work and not under copyright protection in the U.S.;
foreign copyright protection may apply 2022
D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 387–395, 2022.
https://doi.org/10.1007/978-3-030-99524-9_21

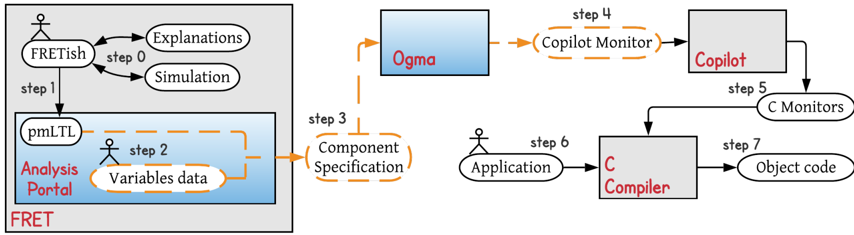

Fig. 1: Step-by-step workflow

capture the semantics of the requirements. We leverage NASA's Formal Requirement Elicitation Tool (FRET) [17, 18] and the runtime monitoring system COPILOT [29, 36, 35]. FRET allows users to express and understand requirements through its intuitive structured natural language (named FRETISH) and elicitation mechanisms, and generates formalizations in temporal logic. COPILOT allows users to specify monitors and compile them to hard real-time C code.

The contribution of this paper is the tight integration of the FRET-COPILOT tools to support the automated synthesis of executable RV monitors directly from requirement specifications. In particular, we present:

- A new tool, named OGMA, that receives requirement formalizations and variable data from FRET and compiles these into COPILOT monitors.
- An extension of the FRET analysis portal to support the generation and export of specifications that can be directly digested by OGMA.
- Preliminary experimental results that evaluate the proposed workflow.

All tools needed by our workflow are available as open source [2, 1, 4].

Related Work. A number of runtime verification languages and systems have been applied in resource-constrained environments [39, 13, 6, 7, 37, 28]. In contrast to our work, these systems do not provide a direct translation from natural language. Several tools [25, 14, 16, 24, 8] formalize natural-language like requirements, but not for the purpose of generating runtime monitors. The STIMULUS tool [22] allows users to express requirements in an extensible, natural-like language that is syntactic sugar for hierarchical state machines. The machines then act as monitors that can be used to validate requirements during the design and testing phases, but are not intended to be used at runtime. FLEA [10] is a formal language for expressing requirements that compiles to runtime monitors in a garbage collected language, making it harder to use in embedded systems; in contrast, our approach generates hard real-time code.

2 Step-by-step Framework Workflow

To integrate FRET and COPILOT, we extended the FRET analysis portal and created the OGMA tool. Figure 1 shows the step-by-step workflow of the complete framework - dashed lines represent the newly added steps (2, 3, and 4). Once requirements are written in FRETISH, FRET helps users understand and refine their requirements through various explanations and simulation (step 0). Next,

NL: "*While flying, if the airspeed is below 100 m/s, the autopilot shall increase the airspeed to at least 100 m/s within 10 seconds.*"

FRETish: `in flight mode if airspeed < 100 the aircraft shall within 10 seconds satisfy (airspeed >= 100)`

pmLTL: H (Lin_flight→(Y ((($O_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → ($O_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) S ((($O_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → ($O_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) & Fin_flight)))) & ((!Lin_flight) S ((!Lin_flight) & Fin_flight)) → ((($O_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → ($O_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) S ((($O_{[=10]}$(((airspeed < 100) & ((Y (!(airspeed < 100))) | Fin_flight)) & (!(airspeed ≥ 100)))) → ($O_{[<10]}$(Fin_flight | (airspeed ≥ 100)))) & Fin_flight)),

where `Fin_flight` (First timepoint in flight mode) is `flight & (FTP | Y !flight)`, `Lin_flight` (Last timepoint in flight mode) is `!flight & Y flight`, `FTP` (First Time Point) is `! Y true`.

Fig. 2: Running example in Natural Language (NL), FRETISH, and pmLTL forms.

FRET automatically translates requirements (step 1) into pure Past-time Metric Linear Temporal Logic (pmLTL) formulas. Next, information about the variables referenced in the requirements must be provided by the user (step 2). The formulas, as well as the provided variables' data, are then combined to generate the Component Specification (step 3). Based on this specification, OGMA creates a complete COPILOT monitor specification (step 4). COPILOT then generates the C Monitor (step 5), which is given along with other C code (step 6) to a C Compiler for the generation (step 7) of the final object code.

Running Example. The next sections illustrate each workflow step using a flight-critical system requirement: airplanes should always avoid stalling (a stall is a sudden loss of lift, which may lead to a loss of control). To avoid stalls, they should fly above a certain speed, known as *stall speed* (as well as stay below a critical angle of attack). Our running requirement example is captured in natural language in Figure 2. For the purposes of this example, we consider the airspeed threshold to be 100 m/s and the correction time to be 10 seconds.

3 FRET Steps

Next we discuss FRET, the requirements tool that constitutes our frontend.

Step 0:* fretish *and semantic nuances. A FRETISH requirement (see running example in Figure 2) contains up to six fields: `scope`, `condition`, `component*`, `shall*`, `timing`, and `response*`. Fields marked with * are mandatory.

`component` specifies the component that the requirement refers to (e.g., aircraft). `shall` expresses that the component's behavior must conform to the requirement. `response` is of the form *satisfy R*, where *R* is a Boolean condition (e.g., satisfy airspeed ≥ 100). `scope` specifies the period when the requirement holds during the execution of the system, e.g., when "in flight mode". `condition`

is a Boolean expression that further constrains when the `response` shall occur (e.g., the requirement becomes relevant only upon airspeed ≤ 100 becoming true). `timing` specifies when the `response` must occur (e.g., within 10 seconds).

Getting a temporal requirement right is usually a tricky task since such requirements are often riddled with semantic subtleties. To help the user, FRET provides a simulator and semantic explanations [17]. For example, the diagram in Figure 3 explains that the requirement is only relevant within the grayed box M (while in flight mode). TC represents the triggering condition (airspeed $<$ 100) and the orange band, with a duration of n=10 seconds, states that the response (airspeed >= 100) is required to hold at least once within the 10 seconds duration, assuming that flight mode holds for at least 10 seconds.

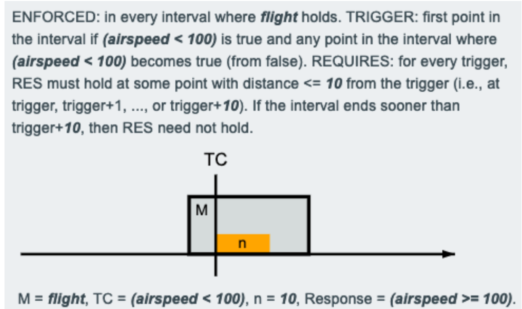

Fig. 3: FRET explanations

Step 1: fretish to pmLTL. For each FRETISH requirement, FRET generates formulas in a variety of formalisms. For the COPILOT integration, we use the generated pmLTL formulas (Figure 2) Clearly, manually writing such formulas can be quite error-prone, while the FRET formalization process has been extensively tested through its *formalization verifier* [17].

Update Variable
FRET Project: TACASpaper
FRET Component: aircraft
FRET Variable: airspeed
Variable Type*: Internal
Data Type*: double
Variable Assignment in Copilot*: altitude >
Parse Errors: mismatched input '<EOF>' expecting {'(', '[', 'constant', 'drop', 'extern', OPOne, 'mux', IDENT}
Lustre CoPilot
CANCEL UPDATE

Fig. 4: FRET variable editor

Steps 2 & 3: Variables data and Component Specification. We extended FRET's analysis portal [3] to capture the information needed to generate Component Specifications for OGMA. To generate a specification, the user must indicate the type (i.e., input, output, internal) and data type (integer, Boolean, double, etc) of each variable (Figure 4).Internal variables represent expressions of input and output variables; if the same expression is used in multiple requirements, an internal variable can be used to substitute it and simplify the requirements. The user must *assign* an expression to each internal variable. In our example, the `flight` internal variable is defined by the expression `altitude > 0.0`, where `altitude` is an input variable. Internal variable assignments can be defined in Lustre [20] or Copilot [29]. Integrated Lustre and Copilot parsers identify parsing errors and return feedback (Figure 4). Once steps 1 and 2 are completed, FRET generates a Component Specification, which contains all requirements in pmLTL and Lustre code, as well as variable data that belong to the same system component.

4 Ogma Steps

Ogma is a command-line tool to produce monitoring applications. Ogma generates monitors in Copilot, and also supports integrating them into larger systems, such as applications built with NASA's core Flight System (cFS) [40].

Step 4: Copilot Monitors. Ogma provides a command `fret-component-spec` to process Component Specifications. The command traverses the Abstract Syntax Tree of the Component Specification, and converts each tree node into its Copilot counterpart. Input and output variables in fret become *extern* streams in Copilot, or time-varying sources of information needed by the monitors:

```
airspeed :: Stream Double
airspeed = extern "airspeed" Nothing
```

Internal variables are also mapped to streams. Each requirement's pmLTL formula is translated into a Boolean stream, paired with a C handler *triggered* when the requirement is violated. In the example below, the property we monitor is associated with a handler, `handlerpropAvoidStall`, which must be implemented separately in C by the user to determine how to address property violations:

```
propAvoidStall :: Stream Bool
propAvoidStall = ((PTLTL.alwaysBeen ((((not (flight)) && ... )))))
spec = trigger "handlerpropAvoidStall" (not propAvoidStall) []
```

5 Copilot Steps

Copilot is a stream-based runtime monitoring language. Copilot streams may contain data of different types. At the top level, specifications consist of pairs of Boolean streams, together with a C handler to be called when the current sample of a stream becomes true. For a detailed introduction to Copilot, see [29].

Step 5: C Monitors. Ogma generates self-contained Copilot monitoring specifications, which can be further compiled into C99 by just compiling and running the Copilot specifications with a Haskell compiler. This process produces two files: a C header and a C implementation.

Step 6: Larger Applications. The C files generated by Copilot are designed to be integrated into larger applications. They provide three connection endpoints: extern variables, a `step` function, and handler functions, which users implement to handle property violations. The code generated has no dynamic memory allocation, loops or recursive calls, it executes in predictable memory and time. For our running example, the header file generated by Copilot declares:

```
extern bool flight;                    extern float airspeed;
void handlerpropAvoidStall(void);      void step(void);
```

Commonly, the calling application will poll sensors, write their values to global variables, call the `step` function, and implement handlers that log property violations or execute corrective actions. Users are responsible for compiling and linking the Copilot code together with their application (step 7).

We also used the running requirement in this paper to monitor a flight in the simulator X-Plane. We wrote an X-Plane plugin to show the state of the C

(a) Cruising (b) Stall (c) Recovery

Fig. 5: Demonstration of Copilot monitor running as X-Plane plugin.

monitor and some additional information on the screen (Fig. 5a). To test the code, we brought an aircraft to a stall by increasing the angle of attack, which also lowered the airspeed (Fig. 5b). After 10 seconds below the specified threshold, the monitor became active, remaining on after executing a stall recovery (Fig. 5c).

6 Preliminary Results

We report on experiments with monitors generated from the publicly available Lockheed Martin Cyber-Physical System (LMCPS) challenge problems [11, 12], which are a set of industrial Simulink model benchmarks and natural language requirements developed by domain experts. LMCPS requirements were previously written in FRETISH [27, 26] by a subset of the authors and were analyzed against the provided models using model checking.

In this paper, we reuse the FRETISH requirements to generate monitors and compare our runtime verification results with the model checking results of [26]. For each Simulink model we generated C code through the automatic code generation feature of Matlab/Simulink. We then attached the generated C monitors to the C code and used the property-based testing system QuickCheck [9] to generate random streams of data, feed them to the system under observation, and report if any of the monitors were activated, based on [30, 31, 34].

We experimented with the Finite State Machine (FSM) and the Control Loop Regulators (REG) LMCPS challenges. For both challenges, our results are consistent with the model checking results - QuickCheck found inputs that activated the monitors, indicating that some requirements were not satisfied. Moreover, it returned results within seconds in cases where model checkers timed out. See [33] for details on the results and [32] for a reproducible artifact.

7 Conclusion

We described an end-to-end framework in which requirements written in structured natural language can be equivalently transformed into monitors and be analyzed against C code. Our framework ensures that requirements and analysis activities are fully aligned: C monitors are derived directly from requirements and not handcrafted. The design of our toolchain facilitates extension with additional front-ends (e.g., JKind Lustre [15]), and backends (e.g., R2U2 [38]). In the future, we plan to explore more use cases, including some from real drone test flights.

References

1. Copilot. `https://github.com/Copilot-Language/copilot/`. Accessed Oct 04, 2021.
2. FRET: Formal Requirements Elicitation Tool. `https://github.com/NASA-SW-VnV/fret/`. Accessed Oct 04, 2021.
3. FRET: Formal Requirements Elicitation Tool - User Manual. `https://github.com/NASA-SW-VnV/fret/blob/master/fret-electron/docs/_media/userManual.md`. See Section "Exporting for Analysis". Accessed Oct 04, 2021.
4. Ogma. `https://github.com/nasa/ogma/`. Accessed Oct 04, 2021.
5. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.
6. J. Baumeister, B. Finkbeiner, S. Schirmer, M. Schwenger, and C. Torens. RT-Lola cleared for take-off: Monitoring autonomous aircraft. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification*, pages 28–39, Cham, 2020. Springer International Publishing.
7. S. Biewer, B. Finkbeiner, H. Hermanns, M. A. Köhl, Y. Schnitzer, and M. Schwenger. RTLola on board: Testing real driving emissions on your phone. In J. F. Groote and K. G. Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–372, Cham, 2021. Springer International Publishing.
8. A. Boteanu, T. Howard, J. Arkin, and H. Kress-Gazit. A model for verifiable grounding and execution of complex natural language instructions. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2649–2654, 2016.
9. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM Sigplan Notices*, 46(4):53–64, 2011.
10. D. Cohen, M. S. Feather, K. Narayanaswamy, and S. S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th International Conference on Software Engineering*, pages 602–603, 1997.
11. C. Elliott. 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 A. F. R. Laboratory, editor, *Safe & Secure Systems and Software Symposium (S5)*, 2015.
12. C. Elliott. An example set of cyber-physical V&V challenges for S5, Lockheed Martin Skunk Works. In A. F. R. Laboratory, editor, *Safe & Secure Systems and Software Symposium (S5)*, 2016.
13. P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, and H. Torfah. StreamLAB: Stream-based monitoring of cyber-physical systems. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification*, pages 421–431, Cham, 2019. Springer International Publishing.
14. A. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, and J. A. Davis. SpeAR v2.0: Formalized past LTL specification and analysis of requirements. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 420–426, 2017.
15. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The jk ind model checker. In *International Conference on Computer Aided Verification*, pages 20–27. Springer, 2018.

16. S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. ARSENAL: automatic requirements specification extraction from natural language. In S. Rayadurgam and O. Tkachuk, editors, *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, volume 9690 of *Lecture Notes in Computer Science*, pages 41–46. Springer, 2016.
17. D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi. Formal requirements elicitation with FRET. In *Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2020)*, 2020.
18. D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann. Automated formalization of structured natural language requirements. *Inf. Softw. Technol.*, 137:106590, 2021.
19. A. Goodloe and L. Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.
20. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
21. K. Havelund and A. Goldberg. *Verify Your Runs*, pages 374–383. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
22. B. Jeannet and F. Gaucher. Debugging embedded systems requirements with STIMULUS: an automotive case-study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, Jan. 2016.
23. J. C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550. ACM, 2002.
24. C. Lignos, V. Raman, C. Finucane, M. Marcus, and H. Kress-Gazit. Provably correct reactive control from natural language. *Auton. Robots*, 38(1):89–105, jan 2015.
25. L. Lúcio, S. Rahman, C.-H. Cheng, and A. Mavin. Just formal enough? Automated analysis of EARS requirements. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 427–434, May 2017.
26. A. Mavridou, H. Bourbouh, P. L. Garoche, and M. Hejase. Evaluation of the FRET and CoCoSim tools on the ten Lockheed Martin cyber-physical challenge problems. Technical Report TM-2019-220374, National Aeronautics and Space Administration, February 2020.
27. 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 *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 300–310, 2020.
28. P. Moosbrugger, K. Y. Rozier, and J. Schumann. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, 51(1):31–61, 2017.
29. I. Perez, F. Dedden, and A. Goodloe. Copilot 3. Technical Report NASA/TM–2020–220587, NASA Langley Research Center, April 2020.
30. I. Perez, A. Goodloe, and W. Edmonson. Fault-tolerant swarms. In *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 47–54. IEEE, 2019.
31. I. Perez and A. E. Goodloe. Fault-tolerant functional reactive programming (extended version). *Journal of Functional Programming*, 30, 2020.

32. I. Perez, A. Mavridou, T. Pressburger, A. Goodloe, and D. Giannakopoulou. Artifact for Automated Translation of Natural Language Requirements to Runtime Monitors. https://doi.org/10.5281/zenodo.5888956. Accessed Jan 21, 2022.
33. I. Perez, A. Mavridou, T. Pressburger, A. Goodloe, and D. Giannakopoulou. Integrating FRET with Copilot: Automated Translation of Natural Language Requirements to Runtime Monitors. Technical Report NASA/TM–20220000049, NASA, January 2022.
34. I. Perez and H. Nilsson. Runtime verification and validation of functional reactive systems. *Journal of Functional Programming*, 30:e28, 2020.
35. L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of the 1st Intl. Conference on Runtime Verification*, LNCS. Springer, November 2010.
36. L. Pike, N. Wegmann, S. Niller, and A. Goodloe. Copilot: Monitoring embedded systems. *Innov. Syst. Softw. Eng.*, 9(4):235–255, Dec. 2013.
37. T. Reinbacher, K. Y. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–372, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
38. J. Schumann, P. Moosbrugger, and K. Y. Rozier. R2u2: monitoring and diagnosis of security threats for unmanned aerial systems. In *Runtime Verification*, pages 233–249. Springer, 2015.
39. H. Torfah. Stream-based monitors for real-time properties. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 91–110. Springer, 2019.
40. J. Wilmot. A core flight software system. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '05, pages 13–14, New York, NY, USA, 2005. ACM.

This is a U.S. government work and not under copyright protection in the U.S.; foreign copyright protection may apply [2022].

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.