EPTCS 329

Proceedings of the

Second Workshop on Formal Methods for Autonomous Systems

Virtual, 7th of December 2020

Edited by: Matt Luckcuck and Marie Farrell

Published: 3rd December 2020
DOI: 10.4204/EPTCS.329
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Autonomous systems are highly complex and present unique challenges for the application of formal methods. Autonomous systems act without human intervention, and are often embedded in a robotic system, so that they can interact with the real world. As such, they exhibit the properties of safety-critical, cyber-physical, hybrid, and real-time systems.

This EPTCS volume contains the proceedings for the second workshop on Formal Methods for Autonomous Systems (FMAS 2020), which was held virtually on the 7th of December 2020. FMAS 2020 was an online, stand-alone workshop, as an adaptation to the ongoing COVID-19 restrictions. Despite the challenges this brought, we aimed to build on the success of the first FMAS workshop, held in 2019.

The goal of FMAS is to bring together leading researchers who are tackling the unique challenges of autonomous systems using formal methods, to present recent and ongoing work. We are interested in the use of formal methods to specify, model, or verify autonomous or robotic systems; in whole or in part. We are also interested in successful industrial applications and potential future directions for this emerging application of formal methods.

FMAS 2020 encouraged the submission of both long and short papers. In total, we received five long papers and one short paper, by authors in Algeria, Canada, India, the United Kingdom, and the United States of America. Each paper received four reviews, and we accepted five papers in total: four long papers and one short paper.

FMAS 2020 hosted two invited speakers. Louise A. Dennis, from the University of Manchester, was invited to talk about verifying machine ethics. This talk focused on the foundations of ethics and how autonomous systems can be made to be explicitly and verifiably ethical. Ivan Perez, from the National Institute of Aerospace and the NASA Formal Methods Group, was invited to talk about Copilot 3, which is a runtime verification framework for real-time embedded systems. This talk described how Copliot synthesises monitor code from a variety of temporal logic specifications.

Despite the disruption caused by the COVID-19 pandemic, our programme committee provided detailed, high quality reviews, which offered useful and constructive feedback to the authors. We thank them for their time and the effort that they have put into their reviews this year, especially given the current, stressful global situation. We would like to thank our invited speakers, Louise A. Dennis and Ivan Perez; the authors who submitted papers; our EPTCS editor, Martin Wirsing; and all of the attendees of FMAS 2020.

Matt Luckcuck **Marie Farrell** **Michael Fisher**

Department of Computer Science, University of Manchester, Manchester, UK

Program Committee

Invited Talk: Verifying Machine Ethics

Louise Dennis
University of Manchester, Manchester, UK

Machine ethics is concerned with the challenge of constructing ethical and ethically behaving artificial agents and systems. One important theme within machine ethics concerns explicitly ethical agents those which are not ethical simply because they are constrained by their programming or deployment to be so but which use a concept of ethics in some way as part of their operation. Normally this requires the provision of rules, utilities or priorities by a programmer, knowledge engineer or user. In this talk I will address the question of how such explicitly ethical programs can be verified. What kind of properties can we consider and what kind of errors might we find?

Invited Talk: Runtime Verification with Copilot 3

Ivan Perez
National Institute of Aerospace
Hampton, Virginia, USA
ivan.perez@nianet.org

Ultra-critical systems require high-level assurance, which cannot always be guaranteed in compile time. The use of runtime verification (RV) [2, 1] enables monitoring these systems in runtime, to detect property violations early and limit their potential consequences. However, the introduction of monitors in ultra-critical systems poses a challenge, as failures and delays in the RV subsystem could affect other subsystems and threaten the mission as a whole.

In this talk we present Copilot 3 [5], a runtime verification framework for real-time embedded systems. Copilot monitors are written in a compositional, stream-based language with support for a variety of temporal logics (e.g., Linear Temporal Logic [7], Past-time Linear Temporal Logic [4], Metric Temporal Logic [3]). Copilot also includes multitude of libraries with functions like, for example, majority vote, used to implement fault-tolerant monitors [6]. All of this which results in robust, high-level specifications that are easier to understand than low-level imperative implementations. The Copilot framework then translates monitor specifications into C99 code with static memory requirements, which can be compiled to run on embedded hardware.

We also discuss how Copilot has been used in experimental research by NIA, NASA and other organizations, its place in relation to other RV frameworks, and possible future directions for RV in autonomous systems.

Acknowledgements This talk is based upon joint work with Alwyn Goodloe and Frank Dedden.

References

[1] Alwyn Goodloe & Lee Pike (2010): *Monitoring Distributed Real-Time Systems: A Survey and Future Directions*. Technical Report NASA/CR-2010-216724, NASA Langley Research Center.

[2] Klaus Havelund & Allen Goldberg (2008): *Verify Your Runs*, pp. 374–383. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-69149-5_40.

[3] Ron Koymans (1990): *Specifying Real-time Properties with Metric Temporal Logic*. Real-Time Syst. 2(4), pp. 255–299, doi:10.1007/BF01995674.

[4] F. Laroussinie, N. Markey & P. Schnoebelen (2002): *Temporal Logic with Forgettable Past*. In: *LICS'02: Proceeding of Logic in Computer Science 2002*, IEEE Computer Society Press, pp. 383–392, doi:10.1109/LICS.2002.1029846.

[5] Ivan Perez, Frank Dedden & Alwyn Goodloe (2020): *Copilot 3*. Technical Memorandum NASA/TM-2020-220587, National Aeronautics and Space Administration, Langley Research Center, Hampton VA 23681-2199, USA.

[6] L. Pike, N. Wegmann, S. Niller & A. Goodloe (2013): *Copilot: monitoring embedded systems*. Innovations in Systems and Software Engineering 9(4), pp. 235–255, doi:10.1007/s11334-013-0223-x.

[7] Amir Pnueli (1977): *The Temporal Logic of Programs*. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, IEEE Computer Society, Washington, DC, USA, pp. 46–57, doi:10.1109/SFCS.1977.32.

How to Formally Model Human in Collaborative Robotics

Mehrnoosh Askarpour
McMaster University
Canada
askarpom@mcmaster.ca

Human-robot collaboration (HRC) is an emerging trend of robotics that promotes the co-presence and cooperation of humans and robots in common workspaces. Physical vicinity and interaction between humans and robots, combined with the uncertainty of human behaviour, could lead to undesired situations where humans are injured. Thus, safety is a priority for HRC applications.

Safety analysis via formal modelling and verification techniques could considerably avoid dangerous consequences, but only if the models of HRC systems are comprehensive and realistic, which requires reasonably realistic models of human behaviour. This paper explores state-of-the-art solutions for modelling human and discusses which ones are suitable for HRC scenarios.

1 Introduction

A new uprising section of robotics is Human-Robot Collaboration (HRC), where human operators are not dealing with robots only through interfaces, but they are physically present in the vicinity of robots, performing hybrid tasks (i.e., partially done by the human and partially by the robot). These applications introduce promising improvements in the industrial manufacturing area by combining human flexibility and machine productivity [52], but they must assure the safety of human operators before being fully deployed to certify that interactions with robots will not cause any harm or injury to humans.

Formal methods have been widely used in robotics for decades in a variety of applications including mission planning [6, 20, 46, 53, 63], formal verification of properties [3, 48, 49, 55], and controllers [8, 24, 66]. They could be an effective means for the safety analysis of traditional and collaborative robotics due to their comprehension and exhaustiveness [32, 77, 78]. However, a formal model of an HRC system should also reflect the human factors that impact the state of the model without dealing with the additional details of human mental and emotional processes. It necessitates building a formal model of human behaviour that replicates their physical presence and the observable manifestation of their behaviour which includes both executing the required job following the expected instructions, and deviations from the expected behaviour (i.e., mistakes, errors, malicious use). Thus, the model of the human for each HRC scenario might be intertwined with the model of the executing job.
A 100% realistic formal model of human might not be a reasonable goal and, just like any other phenomena, any human model is subject to some level of abstraction and simplification. Moreover, unlike robots that perform only a limited set of activities, all possible activities of the human are not foreseeable. Besides, humans are non-deterministic, and a realistic algorithm for their behaviour is not easy to envision.

To tackle these issues, researchers have explored well-established cognitive investigations, task-analytic models, and probabilistic approaches [12]. These three tracks are not mutually exclusive; for example, there are instances of cognitive probabilistic models in the literature that will be discussed later in the paper. The rest of this paper, reports on the state-of-the-art on each of these possibilities and examines their compatibility for HRC scenarios, following a snowballing literature review. Moreover, for each track there are several instances that adhered to the normative human behaviour, while other instances considered erroneous behaviour too; hence, a separate section is dedicated to examples that also model errors.

In addition to the three highlighted possibilities, Bolton et al. [11] considered human-device interface models, which are out of the scope of this paper. They do not consider the physical co-presence of humans and robots and focus only on their remote communication; thus, human physical safety has never been an issue in these studies [10].

The rest of this paper is structured as follows: Section 2 explores the cognitive approaches; Section 3 reviews task analytic approaches; Section 4 discusses the probabilistic techniques; Section 5 specifically discusses the difficulties of modelling human errors; finally Section 6 draws a few conclusions on the best-suited solution for HRC scenarios.

2 Cognitive Models

Cognitive models specify the rationale and knowledge behind human behaviour while working on a set of pre-defined tasks. They are often incorporated in the system models and contain a set of variables that describe the human cognitive state, whose values depend on the state of task execution and the operation environment [12]. Famous examples of well-established cognitive models follow.

SOAR [42] is an extensive cognitive architecture, relying on artificial intelligence principles, that reproduce human reasoning and short and long term memories. However, it only permits one operation at a time which seems not to be realistic in HRC scenarios (e.g., human sends a signal while moving towards the robot).

ACT-R [2] is a detailed and modular architecture that depicts the learning and perception processes of humans. It contains modules that simulate declarative (i.e., known facts like $2+2=4$) and procedural (i.e., knowledge of how to sum two numbers) aspects of human memory. An internal pattern matcher searches for the procedural statement relevant to the task that the human needs to perform (i.e., an entry in declarative memory) at any given time. SAL [38, 45] is an extended version of ACT-R, enriched with a neural architecture called Leabra.

SOAR and ACT-R highlight the cognition behind erroneous behaviours of the human [21] that could impact safety, such as over-trusting or a lack of trust in the system. It could be used to generate realistic models that also reflect a human deviation from the correct instructions. On the other hand, they both lack a formal definition and cannot be directly inputted to automated verification tools. Hence, they must be transformed into a formal model, which is a cumbersome and time-consuming process and requires extensive training for modellers [69]. Moreover, they are detailed and heavy models and, therefore, must be abstracted before formalization to avoid a state-space explosion. There are examples of formal transformation of ACT-R in [15, 30, 43] and of SOAR in [36]. However, they remain dependent on their case-studies or use arbitrary simplifications, and therefore, cannot be re-used as a general approach. Thus, providing a trade-off between abstraction and generality of these two cognitive models is not an easy task.

Programmable User Models (PUM) define a set of goals and actions for humans. The model mirrors both human mental actions (i.e., deciding to pick an object) and physical actions (i.e., pick an object). These models have a notion of a human *mental model* [31, 67] and separate the machine model from the user's perception of it [16], that avoids mode confusions which happen when the observed system behaviour is not the same as the user's expectation [17]. Additionally, Moher et al. [56] assign a *certainty* level to mental models whose different adjustments reveal various human reactions in execution situations. Curzon et al. [23] introduce two customized versions of mental models as naive or experienced.

Since PUM has been around for so long, simplified formal versions of it are defined for a variety of applications such as domestic service robots [72] Formalised with [71], interactive shared applications [18] Formalised with logic formulae, and Air Management System [79] Formalised with Petri-nets. PUM models have a general and accurate semantics and could be well-suited inputs for automated formal verification tools upon simplification. On the other hand, they are very detailed and large and risk the state-space exploration phenomena. As mentioned above, there are simplified versions of PUM models which again are not generic enough to be used for different scenarios including HRC applications. Hence, the effort and time required for customization and simplification remain as high as for SOAR and ACT-R.

3 Task-Analytic Models

Task-analytic models, as their name suggests, analyze human behaviour throughout the execution of the task. Therefore, they study the task as a hierarchy of atomic actions. By definition, these models reflect the expected behaviour of the human, which leads to correct execution of the task, and do not focus on reflecting erroneous behaviour [11]. Recently, however, Bolton et al. [13] put together a task-based taxonomy of erroneous human behaviour that allows errors to be modelled as divergences from task models; Li et al. [47] uses an analytic hierarchy process to identify hazards and increase the efficiency of the executing task.

Examples of task-analytic models follow. Paterno et al. [60] extend ConcurTaskTrees (CTT) [1] to better express the collaboration between multiple human operators in air traffic control; Mitchell and Miller [54] use function models to represent human activities in a simple control system (e.g., system shows information about the current state and operator makes relative control actions); Hartson et al. [33] introduces User Action Notation, a task and user-oriented notation to represent behavioural asynchronous, direct manipulation interface designs; Bolton et al. [14] establishes an Enhanced Operator Function Model, a generic XML-based notation, to gradually decompose tasks into activities, sub-activities and actions.

Task-analytic models depict human-robot co-existence and highlight the active role of humans in the execution of the task. However, they do not always reduce the overall size of the state spaces of the model [10], especially when multiple human operators are involved in the execution. Moreover, they do not offer re-usability (i.e., dependent on the case-study scenario and not generalizable) and generation of erroneous human behaviour.

4 Probabilistic Human Modelling

Another approach to reproduce human non-determinism and uncertainty more vividly are probabilistic models. Unlike deterministic models that produce a single output, stochastic/probabilistic models produce a probability distribution. In theory, a probabilistic human model could be very beneficial for model-based safety assessment in terms of a trade-off between cost and safety. For example, given that human manifests activity A with probability $P << 0.001\%$, and that A causes hazard H which is very expensive to mitigate, system engineers can save some money by not installing expensive mitigation for H that might practically never happen and settle for a more cost-effective mitigation. But we must analyse the challenges of probabilistic models too, so let us discuss a few examples.

Tang et al. [74] propose a bayesian probabilistic human motion model and argue that human mobility behaviour is uncertain but not random, and depends on internal (e.g., individual preferences) and external (e.g., environment) factors. They introduced an algorithm to learn human motion patterns from collected data about human daily activities from GPS and mobile phone data, and extract a probabilistic relation between current human place and past places in an indoor environment. The first issue about this work, and probabilistic models in general, is collecting a big enough dataset for extracting the correct parameters for a probabilistic distribution. The second issue is that the dataset gathered by GPS and cameras is very coarse-grained for modelling delicate HRC situations; they might capture human moving from one corner of a small workspace to another, but do not capture a situation such as human handing an object to the robot gripper or changing the tool-kit installed on the robot arm.

Hawkins et al. [34] present an HRC-oriented approach, based on a probabilistic model of humans and the environment, where the robot infers the current state of the task and performs the appropriate action. This model is very interesting but is more suited for simulation experiments, thus needs adaptive changes to be compatible with formal verification tools. The model also assumes that human performs everything correctly which is not quite realistic.

Tenorth et al. [75] uses Bayesian Logic Networks [37] to create a human model and evaluated their approach with TUM kitchen [58] and the CMU MMAC [76] data sets, both focusing only on full-body movements that exclude many HRC-related activities (e.g., assembling, screw-driving, pick and place, etc).

The Cognitive Reliability and Error Analysis Method (CREAM) [35] is a probabilistic cognitive model that is used for human reliability analysis [26], therefore, focuses on correct and incorrect behaviour both; it defines a set of modes to replicate different types of human behaviour which have different likelihoods to carry out certain errors. Similarly, Systematic Human Error Reduction and Prediction Analysis (SHERPA) [27] is a qualitative probabilistic human error analysis with a task-analytic direction which contains several failure modes: action errors, control errors, recovery errors, communication errors, choice errors. De Felice et al. [28] propose a combination of CREAM and SHERPA that uses empirical data to assign probabilities to each mode; therefore, the results are strongly dependent on the case study domain and the reliability and amount of data.

In recent years, machine-learning algorithms have been used quite extensively to create probabilistic human models [25, 39, 50], however having a reliable and large-enough dataset to learn probabilities from remains an issue. The communities (i.e., providers, users) shall develop such datasets for different domains by storing log histories and integrating them into a unique dataset (e.g., a dataset for industrial assembly tasks, a dataset for service robot tasks, etc.). The larger these datasets become, the more reliable they get, the better the human models extracted from them will be.

5 Human Erroneous Behavior

Human operators are prone to errors—an activity that does not achieve its goal [64], and one significant source of hazards in HRC applications is human errors. Reason [65] metaphorically states that the weaknesses of safety procedures in a system allow for the occurrence of human errors. Hence, a realistic human model for safety analysis shall be able to replicate errors, too.
The previous sections explained different possibilities to model human behaviour. They all could be used to model human errors, too, but are not always used as such; some of the papers discussed above, consider errors and others entirely skip them. This challenge is discussed in a separate section, but all of the papers discussed below could be listed in at least one of the previous three sections.

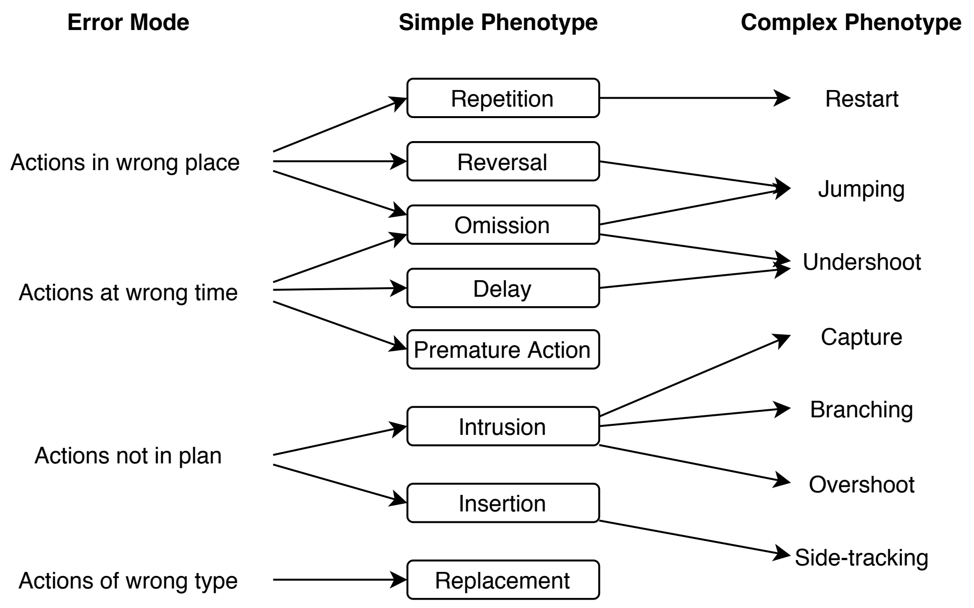


Figure (1) A taxonomy of erroneous actions phenotypes, taken from [29]. Wrong place refers to the action's temporal position in the execution sequence, not to a location in the layout; undershooting, which occurs when the action stops too early; sidetracking occurs when a segment of unrelated action is carried out, then the correct sequence is resumed; capturing occurs where an unrelated action sequence is carried out instead of the expected one; branching is where the wrong sequence of actions is chosen; overshooting happens where the action carries on past its correct endpoint by not recognizing its post-conditions.

One must first recognize errors and settle for a precise definition for them to model them. Although there is no widely used classification of human errors, some notable references follow.

Reason [64] classifies errors as behavioral (i.e., task-related factors), contextual (i.e., environmental factors) and conceptual (i.e., human cognition). Shin et al. [70] divide errors into two main groups; location errors happen when humans shall find a specific location for the task, and orientation errors happen due to humans' various timing or modality to perform a task. Hollnagel [35] classifies human errors in eight simple phenotypes: repeating, reversing the order, omission, late or early execution, replacement, intrusion of actions and inserting an additional action from elsewhere in the task; the paper then combines them to get complex phenotypes, as shown in Figure 1, and manually introduces them in the formal model of the model which produces too many false negatives. This issue might be resolved by introducing a probability distribution on phenotypes.

Kirwan has conducted an extensive review on human error identification techniques in [41] and suggested the following as the most frequent version:

- Slips and lapses regarding the quality of performance.
- Cognitive, diagnostic and decision-making errors due to misunderstanding the instructions.
- Maintenance errors and latent failures during maintenance activities.
- Errors of commission when the human does an incorrect or irrelevant activity.
- Idiosyncratic errors regarding social variables and human emotional state.
- Software programming errors leading to malfunctioning controllers that put the human in danger.

Cerone et al. [19] uses temporal logic to model human errors that are defined as: fail to observe potential conflict, ineffective or no response to observed conflict, fail to detect the criticality of the conflict.

As explained above, the error definition is context-dependent and might vary from one domain to another. It seems that the taxonomy presented in [29] is comprehensive enough for possibilities in HRC scenarios. Nevertheless, the discussion above was a general description of the violation of Norms; we must explore also papers that model the above definitions.

Many of the works on modelling human errors are inspired by one of the two main steps of human reliability analysis [26] that are error identification and error probability quantification [57]. For example, Martinie et al. [51] proposes a deterministic task-analytic approach to identify errors, while CREAM, HEART [80], THERP [73] and THEA [62] are probabilistic methods for modelling human. However, the models generated by these methods need formalization and serious customization because they strongly depend on their case study. Examples of other works with the same issues follow.

[9, 59] study the impacts of miscommunication between multiple human operators while interacting with critical systems; [7, 61] explore human deviation from correct instructions using ConcurTaskTrees; [68] upgrades the SAL cognitive model with systematic errors taken from empirical data.

On the other hand, several works developed formal human models. For example, Curzon and Blandford [22] propose a cognitive formal model in higher-order logic that separates user-centred and machine-centred concerns. It uses the HOL proof system [44] to prove that following the proper design rules, the machine-centred model does not allow for errors in the user-centred model. They experimented with including strong enough design rules in the model, so to exclude cognitively plausible errors by design [21]. This method clearly ignores realistic scenarios in which human actually performs an error. In another example, Kim et al. [40] map human non-determinism to a finite state automaton. However, their model is limited to the context of prospective control. Askarpour et al. [4,5] formalize simple phenotypes introduced by Fields [29] with temporal logic, and integrate the result in the overall system model which is verified against a physical safety property.

6 Conclusions

This paper reviews the state-of-the-art on modelling human in HRC applications for safety analysis. The paper explored several papers, excluding those on interface applications, and grouped them into three main groups which are not mutually exclusive: cognitive, task-analytic, and probabilistic models. Then it reviewed state-of-the-art on modelling human errors and mistakes which is absolutely necessary to be considered for safety. Human error models expand over the three main groups but have been discussed separately for more clarity. Table 1 summarizes the observations from which the following conclusions are drawn:

- Cognitive models are very detailed and extensive. They often originate from different research areas (e.g., psychology), therefore have been developed with a different mentality from that of formal methods practitioners. So, they must be Formalised; but their formalization requires a lot of effort for abstraction and HRC-oriented customization. They also require specialist training for modellers to understand the models and be able to modify them.
- Among cognitive models, PUM seems to be the one with more available formal instances. It also could be tailored to different scenarios with much less time, effort and training, compared to ACT-R and SOAR.

Table (1) A summary of the discussed papers, extracted by a snowballing literature review. × means that the feature is not included, ✓means that the feature is included, ? means that the feature has not been explored by the papers of the row but potentially could be addressed by their proposed approach (regardless of the required effort and the resulting efficiency), "semi" means that the model has partially formal semantics but is not in a form to be fed to an automated verification tool. When a reproducible model or approach has been used, the name of the approach is mentioned instead of a ✓. The acronyms used here are explained in Table 2.

modelling Approach	Cognitive	Task-Analytic	Probabilistic	Modelling Errors	Formalised	HRC Compatible
[42]	SOAR	×	×	?	×	×
[2]	ACT-R	×	×	?	×	×
[15,30]	ACT-R	×	×	?	✓	×
[36]	SOAR	×	×	?	✓	×
[31,67,72]	PUM	×	×	?	✓	?
[18,79]	PUM	×	×	?	✓	×
[13]	×	EOFM	×	✓	×	✓
[47]	×	Graphs	×	×	×	×
[60]	×	CCT	×	?	semi	×
[54]	×	FSA	×	✓	✓	×
[33]	×	UAN	×	×	semi	×
[14]	×	EOFM	×	×	×	×
[74]	×	×	Bayesian	✓	×	×
[34]	×	×	DBN	×	×	✓
[75]	×	×	BLN	?	×	×
[28]	×	×	CREAM and SHERPA	×	×	?
[25,39,50]	×	×	ML	?	×	?
[29]	✓	✓	×	✓	×	?
[19]	×	OCM	×	✓	✓	?
[51]	×	✓	×	✓	✓	×
[80]	×	✓	CREAM and HEART	✓	✓	?
[73]	×	✓	THERP	✓	✓	?
[62]	×	✓	THEA	✓	✓	?
[59]	✓	EOFMC	×	✓	LTL	×
[9]	SAL	EOFM	×	✓	×	×
[7]	×	CCT	×	✓	semi	?
[61]	×	CCT	×	✓	×	?
[68]	SAL	×	×	✓	×	×
[22]	PUM	✓	×	✓	Higher-Order Logic	×
[21]	PUM	✓	×	✓	Temporal Logic	?
[40]	[26]	×	×	✓	FSA	×
[5]	×	✓	×	✓	Temporal Logic	✓

Table (2) Acronyms Explained.

PUM	Programmable User Model
OCM	Operator Choice Model
EOFM	Enhanced Operator Function Model
EOFMC	Enhanced Operator Function Model with Communications
UAN	User Action Notation
CCT	ConcurTaskTrees
FSA	Finite State Automaton
CREAM	The Cognitive Reliability and Error Analysis Method
SHERPA	Systematic Human Error Reduction and Prediction Analysis
LTL	Linear Temporal Logic
DBN	Dynamic Bayes Network
BLN	Bayesian Logic Networks
ML	Machine Learning
THERP	Technique for Human Error-rate Prediction
THEA	Technique for Human Error Assessment Early in Design
HEART	Human error assessment and reduction technique
SHERPA	Systematic Human Error Reduction and Prediction Analysis

- Task-analytic models offer little reusability; they are so intertwined with the task definition that a slight change in the task might cause significant changes to the model. They also must be first defined with a hierarchical notation for the easy and clear decomposition of the task and then be translated to an understandable format for a verification tool.
- Probabilistic models seem to be an optimal solution; they combine either task-analytic or cognitive models with probability distributions. However, the biggest issue here is to have reliable and large-enough data sets to extract the parameters of probability distributions from. It requires the robotics community to produce huge training datasets from their system history logs (e.g., how frequently human moves in the workspace, the average value of human velocity while performing a specific task, the number of human interruptions during the execution, the number of emergency stops, the number of reported errors in an hour of execution, ...).
- None of the three approaches above is enough if they exclude human errors. There are several works on outlining and classifying human errors. The best way to have a unified terminology would be for standards organizations in each domain to introduce a list of the most frequent human errors in that domain. It would be possible only if the datasets mentioned above are available. Modelling all of the possible human errors might not be feasible, but modelling those that occur more often is feasible and considerably improves the quality of the final human model.
- Error models might introduce a huge amount of false positive cases (i.e., incorrectly reporting a hazard when it is safe) during safety analysis. Thus, probabilistic error models might be the best combination to resolve it.
- In the safety analysis of HRC systems, the observable behaviour of humans and its consequences (i.e., how it impacts the state of the system) are very important. However, the cognitive elements behind it are not really valuable to the analysis. One could use them as a black-box that derives the observable behaviour with a certain probability but the details of what happens inside the box

do not really matter. Therefore, the cognitive model seems to contain a huge amount of detail that does not necessarily add value to the safety analysis but makes the model heavy and the verification long.

- Table 1 suggests that a combination of probabilistic and task-analytic approaches that model erroneous human behaviour is the best answer for a formal verification method for safety analysis of HRC systems.

References

[1] (2012): *Concur Task Trees (CTT)*. available from `w3.org`. Accessed: 2017-07-06.

[2] John R Anderson (1996): *ACT: A simple theory of complex cognition.* American Psychologist 51, doi:10.1037/0003-066X.51.4.355.

[3] Mehrnoosh Askarpour (2016): *Risk Assessment in Collaborative Robotics*. In: Formal Methods Doctoral Symposium, FM-DS, CEUR-WS 1744.

[4] Mehrnoosh Askarpour, Dino Mandrioli, Matteo Rossi & Federico Vicentini (2017): *Modeling Operator Behavior in the Safety Analysis of Collaborative Robotic Applications*. In Stefano Tonetta, Erwin Schoitsch & Friedemann Bitsch, editors: Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15, 2017, Proceedings, Lecture Notes in Computer Science 10488, Springer, pp. 89–104, doi:10.1007/978-3-319-66266-4_6.

[5] Mehrnoosh Askarpour, Dino Mandrioli, Matteo Rossi & Federico Vicentini (2019): *Formal model of human erroneous behavior for safety analysis in collaborative robotics*. Robotics and Computer-Integrated Manufacturing 57, pp. 465 – 476, doi:10.1016/j.rcim.2019.01.001.

[6] Mehrnoosh Askarpour, Claudio Menghi, Gabriele Belli, Marcello M. Bersani & Patrizio Pelliccione (2020): *Mind the gap: Robotic Mission Planning Meets Software Engineering*. In: FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020, pp. 55–65, doi:10.1145/3372020.3391561.

[7] Sandra Basnyat & Philippe Palanque (2005): *A task pattern approach to incorporate user deviation in task models*. In: Proc. 1st ADVISES Young Researchers Workshop. Liege, Belgium.

[8] Marcello M. Bersani, Matteo Soldo, Claudio Menghi, Patrizio Pelliccione & Matteo Rossi (2020): *PuRSUE -from specification of robotic environments to synthesis of controllers*. Formal Aspects Comput. 32(2), pp. 187–227, doi:10.1109/TAC.2011.2176409.

[9] Matthew L. Bolton (2015): *Model Checking Human-Human Communication Protocols Using Task Models and Miscommunication Generation*. J. Aerospace Inf. Sys. 12(7), pp. 476–489, doi:10.1177/1555343413490944.

[10] Matthew L. Bolton & Ellen J. Bass (2010): *Formally verifying human–automation interaction as part of a system model: limitations and tradeoffs*. Innovations in Systems and Software Engineering 6(3), pp. 219–231, doi:10.1007/s11334-010-0129-9.

[11] Matthew L. Bolton, Ellen J. Bass & Radu I. Siminiceanu (2012): *Generating Phenotypical Erroneous Human Behavior to Evaluate Human-automation Interaction Using Model Checking*. Int. J. Hum.-Comput. Stud. 70(11), pp. 888–906, doi:10.1016/j.ijhcs.2012.05.010.

[12] Matthew L. Bolton, Ellen J. Bass & Radu I. Siminiceanu (2013): *Using Formal Verification to Evaluate Human-Automation Interaction: A Review*. IEEE Trans. Systems, Man, and Cybernetics: Systems 43(3), pp. 488–503, doi:10.1109/TSMCA.2012.2210406.

[13] Matthew L. Bolton, Kylie Molinaro & Adam Houser (2019): *A formal method for assessing the impact of task-based erroneous human behavior on system safety*. Reliab. Eng. Syst. Saf. 188, pp. 168–180. Available at `https://doi.org/10.1016/j.ress.2019.03.010`.

[14] Matthew L. Bolton, Radu I. Siminiceanu & Ellen J. Bass (2011): *A Systematic Approach to Model Checking Human-Automation Interaction Using Task Analytic Models.* *IEEE Trans. Systems, Man, and Cybernetics, Part A* 41(5), pp. 961–976, doi:10.1109/TSMCA.2011.2109709.

[15] Fiemke Both & Annerieke Heuvelink (2007): *From a formal cognitive task model to an implemented ACT-R model.* In: *Proceedings of the 8th International Conference on Cognitive Modeling (ICCM)*, pp. 199–204.

[16] Jan Bredereke & Axel Lankenau (2002): *A Rigorous View of Mode Confusion.* In: *Proceedings of the 21st International Conference on Computer Safety, Reliability and Security*, SAFECOMP '02, Springer-Verlag, London, UK, UK, pp. 19–31, doi:10.1016/0167-6423(95)96871-J.

[17] Jan Bredereke & Axel Lankenau (2005): *Safety-relevant mode confusions modelling and reducing them.* *Reliability Engineering & System Safety* 88(3), pp. 229 – 245, doi:10.1016/j.ress.2004.07.020.

[18] Richard Butterworth, Ann Blandford & David Duke (2000): *Demonstrating the Cognitive Plausibility of Interactive System Specifications.* *Formal Aspects of Computing* 12(4), pp. 237–259, doi:10.1007/s001650070021.

[19] Antonio Cerone, Peter A. Lindsay & Simon Connelly (2005): *Formal Analysis of Human-computer Interaction using Model-checking.* In Bernhard K. Aichernig & Bernhard Beckert, editors: *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*, IEEE Computer Society, pp. 352–362, doi:10.1109/SEFM.2005.19. Available at `https://ieeexplore.ieee.org/xpl/conhome/10529/proceeding`.

[20] Matthew Crosby, Ronald P. A. Petrick, Francesco Rovida & Volker Krüger (2017): *Integrating Mission and Task Planning in an Industrial Robotics Framework.* In Laura Barbulescu, Jeremy Frank, Mausam & Stephen F. Smith, editors: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, AAAI Press, pp. 471–479. Available at `https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15715`.

[21] Paul Curzon & Ann Blandford (2002): *From a Formal User Model to Design Rules.* In: *Interactive Systems. Design, Specification, and Verification, 9th International Workshop, DSV-IS 2002, Rostock Germany, June 12-14, 2002*, pp. 1–15, doi:10.1007/3-540-36235-5_1.

[22] Paul Curzon & Ann Blandford (2004): *Formally justifying user-centred design rules: a case study on post-completion errors.* In: *International Conference on Integrated Formal Methods*, Springer, pp. 461–480, doi:10.1007/3-540-47884-1_12.

[23] Paul Curzon, Rimvydas Rukšėnas & Ann Blandford (2007): *An approach to formal verification of human–computer interaction.* *Formal Aspects of Computing* 19(4), pp. 513–550, doi:10.1007/s00165-007-0035-6.

[24] Lavindra de Silva, Paolo Felli, David Sanderson, Jack C. Chaplin, Brian Logan & Svetan Ratchev (2019): *Synthesising process controllers from formal models of transformable assembly systems.* *Robotics and Computer-Integrated Manufacturing* 58, pp. 130 – 144, doi:10.1016/j.rcim.2019.01.014.

[25] R. Dillmann, O. Rogalla, M. Ehrenmann, R. Zöliner & M. Bordegoni (2000): *Learning Robot Behaviour and Skills Based on Human Demonstration and Advice: The Machine Learning Paradigm.* In John M. Hollerbach & Daniel E. Koditschek, editors: *Robotics Research*, Springer London, London, pp. 229–238, doi:10.1007/978-1-4471-1555-7.

[26] E.M. Dougherty & J.R. Fragola (1988): *Human reliability analysis.* New York, NY; John Wiley and Sons Inc.

[27] DE Embrey (1986): *SHERPA: A systematic human error reduction and prediction approach.* In: *Proceedings of the international topical meeting on advances in human factors in nuclear power systems.*

[28] F. De Felice, A. Petrillo & F. Zomparelli (2016): *A Hybrid Model for Human Error Probability Analysis.* *IFAC-PapersOnLine* 49(12), pp. 1673 – 1678, doi:10.1016/j.ifacol.2016.07.821. 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016.

[29] Robert E. Fields (2001): *Analysis of erroneous actions in the design of critical systems.* Ph.D. thesis, University of York.

[30] Daniel Gall & Thom W. Frühwirth (2014): *A Formal Semantics for the Cognitive Architecture ACT-R*. In: Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers, pp. 74–91, doi:10.1007/978-3-319-17822-6_5.

[31] Wayne D. Gray (2000): *The Nature and Processing of Errors in Interactive Behavior*. Cognitive Science 24(2), pp. 205–248, doi:10.1207/s15516709cog2402_2.

[32] Jrmie Guiochet, Mathilde Machin & Hlne Waeselynck (2017): *Safety-critical advanced robots: A survey*. Robotics and Autonomous Systems 94, pp. 43 – 52, doi:10.1016/j.robot.2017.04.004.

[33] H. Rex Hartson, Antonio C. Siochi & D. Hix (1990): *The UAN: A User-oriented Representation for Direct Manipulation Interface Designs*. ACM Trans. Inf. Syst. 8(3), pp. 181–203, doi:10.1145/964967.801131.

[34] K. P. Hawkins, Nam Vo, S. Bansal & A. F. Bobick (2013): *Probabilistic human action prediction and wait-sensitive planning for responsive human-robot collaboration*. In: 2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids), pp. 499–506, doi:10.1109/HUMANOIDS.2013.7030020.

[35] Erik Hollnagel (1998): *Cognitive reliability and error analysis method (CREAM)*. Elsevier.

[36] Andrew Howes & Richard M. Young (1997): *The Role of Cognitive Architecture in Modeling the User: Soar's Learning Mechanism*. Human-Computer Interaction 12(4), pp. 311–343, doi:10.1007/BF01099424.

[37] Dominik Jain, Stefan Waldherr & Michael Beetz (2011): *Bayesian Logic Networks*.

[38] David J. Jilk, Christian Lebiere, Randall C. O'Reilly & John R. Anderson (2008): *SAL: an explicitly pluralistic cognitive architecture*. J. Exp. Theor. Artif. Intell. 20(3), pp. 197–218, doi:10.1007/10719871.

[39] Been Kim (2015): *Interactive and interpretable machine learning models for human machine collaboration*. Ph.D. thesis, Massachusetts Institute of Technology.

[40] Namhun Kim, Ling Rothrock, Jaekoo Joo & Richard A. Wysk (2010): *An affordance-based formalism for modeling human-involvement in complex systems for prospective control*. In: Proceedings of the 2010 Winter Simulation Conference, WSC 2010, Baltimore, Maryland, USA, 5-8 December 2010, IEEE, pp. 811–823, doi:10.1109/WSC.2010.5679107. Available at `https://ieeexplore.ieee.org/xpl/conhome/5672636/proceeding`.

[41] Barry Kirwan (1998): *Human error identification techniques for risk assessment of high risk systems-Part 1: review and evaluation of techniques*. Applied Ergonomics 29(3), pp. 157 – 177, doi:10.1016/S0003-6870(98)00010-6.

[42] John E Laird (2012): *The Soar cognitive architecture*. MIT Press, doi:10.7551/mitpress/7688.001.0001.

[43] Vincent Langenfeld, Bernd Westphal & Andreas Podelski (2019): *On Formal Verification of ACT-R Architectures and Models*. In Ashok K. Goel, Colleen M. Seifert & Christian Freksa, editors: Proceedings of the 41th Annual Meeting of the Cognitive Science Society, CogSci 2019: Creativity + Cognition + Computation, Montreal, Canada, July 24-27, 2019, cognitivesciencesociety.org, pp. 618–624. Available at `https://mindmodeling.org/cogsci2019/papers/0124/index.html`.

[44] Kung-Kiu Lau & Mario Ornaghi (1995): *Towards an Object-Oriented Methodology for Deductive Synthesis of Logic Programs*. In: Logic Programming Synthesis and Transformation, 5th International Workshop, LOPSTR'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings, pp. 152–169, doi:10.1007/3-540-60939-3_11.

[45] Christian Lebiere, Randall C. O'Reilly, David J. Jilk, Niels Taatgen & John R. Anderson (2008): *The SAL Integrated Cognitive Architecture*. In: Biologically Inspired Cognitive Architectures, Papers from the 2008 AAAI Fall Symposium, Arlington, Virginia, USA, November 7-9, 2008, AAAI Technical Report FS-08-04, AAAI, pp. 98–104. Available at `http://www.aaai.org/Library/Symposia/Fall/2008/fs08-04-027.php`.

[46] B. Li, B. R. Page, B. Moridian & N. Mahmoudian (2020): *Collaborative Mission Planning for Long-Term Operation Considering Energy Limitations*. IEEE Robotics and Automation Letters 5(3), pp. 4751–4758, doi:10.1109/LRA.2020.3003881.

[47] Meng kun Li, Yong kuo Liu, Min jun Peng, Chun li Xie & Li qun Yang (2016): *The decision making method of task arrangement based on analytic hierarchy process for nuclear safety in radiation field.* Progress in Nuclear Energy 93, pp. 318 – 326.

[48] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon & Michael Fisher (2019): *Formal Specification and Verification of Autonomous Robotic Systems: A Survey.* ACM Comput. Surv. 52(5), pp. 100:1–100:41, doi:10.1007/s10458-010-9146-1.

[49] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon & Michael Fisher (2019): *A Summary of Formal Specification and Verification of Autonomous Robotic Systems.* In Wolfgang Ahrendt & Silvia Lizeth Tapia Tarifa, editors: Integrated Formal Methods - 15th International Conference, IFM, Lecture Notes in Computer Science 11918, Springer, pp. 538–541, doi:10.1145/1592434.1592436.

[50] Andrea Mannini & Angelo Maria Sabatini (2010): *Machine Learning Methods for Classifying Human Physical Activity from On-Body Accelerometers.* Sensors 10(2), pp. 1154–1175, doi:10.3390/s100201154.

[51] Célia Martinie, Philippe A. Palanque, Racim Fahssi, Jean-Paul Blanquart, Camille Fayollas & Christel Seguin (2016): *Task Model-Based Systematic Analysis of Both System Failures and Human Errors.* IEEE Trans. Human-Machine Systems 46(2), pp. 243–254, doi:10.1109/THMS.2014.2365956.

[52] Björn Matthias (2015): *Risk Assessment for Human-Robot Collaborative Applications.* In: Workshop IROS - Physical Human-Robot Collaboration: Safety, Control, Learning and Applications.

[53] Claudio Menghi, Sergio Garcia, Patrizio Pelliccione & Jana Tumova (2018): *Multi-robot LTL Planning Under Uncertainty.* In Klaus Havelund, Jan Peleska, Bill Roscoe & Erik de Vink, editors: Formal Methods, Springer International Publishing, Cham, pp. 399–417, doi:10.1177/0278364915595278.

[54] Christine M. Mitchell & R. A. Miller (1986): *A Discrete Control Model of Operator Function: A Methodology for Information Display Design.* IEEE Trans. Systems, Man, and Cybernetics 16(3), pp. 343–357, doi:10.1109/TSMC.1986.4308966.

[55] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis & Jim Woodcock (2019): *RoboChart: modelling and verification of the functional behaviour of robotic applications.* Software and Systems Modeling 18(5), pp. 3097–3149, doi:10.1145/197320.197322.

[56] Victor Moher, Thomas G.and Dirda (1995): *Revising mental models to accommodate expectation failures in human-computer dialogues*, pp. 76–92. Springer Vienna, Vienna.

[57] A. Mosleh & Y.H. Chang (2004): *Model-based human reliability analysis: prospects and requirements.* Reliability Engineering & System Safety 83(2), pp. 241 – 253, doi:10.1016/j.ress.2003.09.014. Human Reliability Analysis: Data Issues and Errors of Commission.

[58] tum technische universitat munchen (2012): *The TUM Kitchen Data Set.* Available at `https://ias.in.tum.de/dokuwiki/software/kitchen-activity-data`.

[59] Dan Pan & Matthew L. Bolton (2016): *Properties for formally assessing the performance level of human-human collaborative procedures with miscommunications and erroneous human behavior.* International Journal of Industrial Ergonomics, pp. –.

[60] Fabio Paternò, Cristiano Mancini & Silvia Meniconi (1997): *ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models.* In Steve Howard, Judy Hammond & Gitte Lindgaard, editors: Human-Computer Interaction, INTERACT '97, IFIP TC13 Interantional Conference on Human-Computer Interaction, 14th-18th July 1997, Sydney, Australia, IFIP Conference Proceedings 96, Chapman & Hall, pp. 362–369.

[61] Fabio Paternò & Carmen Santoro (2002): *Preventing user errors by systematic analysis of deviations from the system task model.* International Journal of Human-Computer Studies 56(2), pp. 225–245, doi:10.1006/ijhc.2001.0523.

[62] Steven Pocock, Michael D. Harrison, Peter C. Wright & Paul Johnson (2001): *THEA: A Technique for Human Error Assessment Early in Design.* In Michitaka Hirose, editor: Human-Computer Interaction INTERACT '01: IFIP TC13 International Conference on Human-Computer Interaction, Tokyo, Japan, July 9-13, 2001, IOS Press, pp. 247–254.

[63] Gayathri R. & V. Uma (2018): *Ontology based knowledge representation technique, domain modeling languages and planners for robotic path planning: A survey*. ICT Express 4(2), pp. 69 – 74, doi:10.1016/j.icte.2018.04.008. SI on Artificial Intelligence and Machine Learning.

[64] James Reason (1990): *Human error*. Cambridge university press, doi:10.1017/CBO9781139062367.

[65] James Reason (2000): *Human error: models and management*. BMJ: British Medical Journal 320(7237), p. 768, doi:10.1136/bmj.320.7237.768.

[66] Pedro Ribeiro, Alvaro Miyazawa, Wei Li, Ana Cavalcanti & Jon Timmis (2017): *Modelling and Verification of Timed Robotic Controllers*. In Nadia Polikarpova & Steve Schneider, editors: Integrated Formal Methods, Springer International Publishing, Cham, pp. 18–33, doi:10.1007/BFb0020949.

[67] FRANK E. RITTER & RICHARD M. YOUNG (2001): *Embodied models as simulated users: introduction to this special issue on using cognitive models to improve interface design*. International Journal of Human-Computer Studies 55(1), pp. 1 – 14, doi:10.1006/ijhc.2001.0471.

[68] Rimvydas Ruksenas, Jonathan Back, Paul Curzon & Ann Blandford (2009): *Verification-guided modelling of salience and cognitive load*. Formal Asp. Comput. 21(6), pp. 541–569, doi:10.1007/s00165-008-0102-7.

[69] Dario D. Salvucci & Frank J. Lee (2003): *Simple cognitive modeling in a complex cognitive architecture*. In Gilbert Cockton & Panu Korhonen, editors: Proceedings of the 2003 Conference on Human Factors in Computing Systems, CHI 2003, Ft. Lauderdale, Florida, USA, April 5-10, 2003, ACM, pp. 265–272, doi:10.1145/642611.642658.

[70] Dongmin Shin, Richard A. Wysk & Ling Rothrock (2006): *Formal model of human material-handling tasks for control of manufacturing systems*. IEEE Trans. Systems, Man, and Cybernetics, Part A 36(4), pp. 685–696, doi:10.1109/TSMCA.2005.853490.

[71] Maarten Sierhuis et al. (2001): *Modeling and simulating work practice: BRAHMS: a multiagent modeling and simulation language for work system analysis and design*.

[72] Richard Stocker, Louise Dennis, Clare Dixon & Michael Fisher (2012): *Verifying Brahms Human-Robot Teamwork Models*. In: Logics in Artificial Intelligence: 13th European Conference, JELIA, doi:10.1023/A:1022920129859.

[73] Alan D Swain & Henry E Guttmann (1983): *Handbook of human-reliability analysis with emphasis on nuclear power plant applications. Final report*. Technical Report, Sandia National Labs., Albuquerque, NM (USA).

[74] Bo Tang, Chao Jiang, Haibo He & Yi Guo (2016): *Probabilistic human mobility model in indoor environment*, pp. 1601–1608. doi:10.1109/IJCNN.2016.7727389. Available at `https://ieeexplore.ieee.org/xpl/conhome/7593175/proceeding`.

[75] Moritz Tenorth, Fernando De la Torre & Michael Beetz (2013): *Learning probability distributions over partially-ordered human everyday activities*. In: 2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany, May 6-10, 2013, IEEE, pp. 4539–4544, doi:10.1109/ICRA.2013.6631222. Available at `https://ieeexplore.ieee.org/xpl/conhome/6615630/proceeding`.

[76] Fernando de la Torre, Jessica K. Hodgins, Javier Montano & Sergio Valcarcel (2009): *Detailed Human Data Acquisition of Kitchen Activities: the CMU-Multimodal Activity Database (CMU-MMAC)*. In: CHI 2009 Workshop. Developing Shared Home Behavior Datasets to Advance HCI and Ubiquitous Computing Research.

[77] Federico Vicentini, Mehrnoosh Askarpour, Matteo Rossi & Dino Mandrioli (2020): *Safety Assessment of Collaborative Robotics Through Automated Formal Verification*. IEEE Trans. Robotics 36(1), pp. 42–61, doi:10.1109/TRO.2019.2937471.

[78] M.L. Visinsky, J.R. Cavallaro & I.D. Walker (1994): *Robotic fault detection and fault tolerance: A survey*. Reliability Engineering & System Safety 46(2), pp. 139 – 158, doi:10.1016/0951-8320(94)90132-5.

[79] Bernd Werther & Eckehard Schnieder (2005): *Formal Cognitive Resource Model: Modeling of human behavior in complex work environments*. In: 2005 International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA 2005), International Conference on Intelligent Agents,

Web Technologies and Internet Commerce (IAWTIC 2005), 28-30 November 2005, Vienna, Austria, IEEE Computer Society, pp. 606–611, doi:10.1109/CIMCA.2005.1631535. Available at `https://ieeexplore.ieee.org/xpl/conhome/10869/proceeding`.

[80] J. C. Williams (1988): *A data-based method for assessing and reducing human error to improve operational performance*. In: *Conference Record for 1988 IEEE Fourth Conference on Human Factors and Power Plants,*, pp. 436–450, doi:10.1109/HFPP.1988.27540.

Towards Compositional Verification for Modular Robotic Systems [*]

Rafael C. Cardoso Louise A. Dennis Marie Farrell Michael Fisher Matt Luckcuck
Department of Computer Science
The University of Manchester
Manchester, United Kingdom
{rafael.cardoso, louise.dennis, marie.farrell, michael.fisher, matthew.luckcuck}@manchester.ac.uk

Software engineering of modular robotic systems is a challenging task, however, verifying that the developed components all behave as they should individually and as a whole presents its own unique set of challenges. In particular, distinct components in a modular robotic system often require different verification techniques to ensure that they behave as expected. Ensuring whole system consistency when individual components are verified using a variety of techniques and formalisms is difficult. This paper discusses how to use compositional verification to integrate the various verification techniques that are applied to modular robotic software, using a First-Order Logic (FOL) contract that captures each component's assumptions and guarantees. These contracts can then be used to guide the verification of the individual components, be it by testing or the use of a formal method. We provide an illustrative example of an autonomous robot used in remote inspection. We also discuss a way of defining confidence for the verification associated with each component.

1 Introduction

Autonomous robotic systems are being used more frequently in safety-critical scenarios. Examples include monitoring offshore structures [16], nuclear inspection and decommissioning [4], and space exploration [31]. Ensuring that the software which controls the robot behaves as it should is crucial, particularly as modern robotic systems become more modular, and are deployed alongside humans in both safety- and mission-critical scenarios.

Robotic software is often developed using a middleware to facilitate interoperability, such as ROS[1] or GenoM[2]. They share some abstract concepts [26], namely that systems are composed of communicating components. The approach that we describe in this paper is not restricted to any particular robotic middleware, instead it can be applied to a range of modular systems.

Distinct components in a modular system often require different verification techniques, ranging from software testing to formal methods. In fact, *integrating* (formal and non-formal) verification techniques is crucial particularly for robotic applications [14]. It is essential for the verification to be carried out using the most suitable technique or formalism for each component. However, performing compositional verification via linking heterogeneous verification results of individual components is difficult and the current state-of-the-art for robotic software development does not provide an easy way of achieving this.

Our approach is to construct a high-level First-Order Logic (FOL) contract specification of the system. The FOL contract describes the expected input, required output, and Assume-Guarantee [19] conditions for each component. This abstract specification can be seen as a logical prototype for individual

[*]Work supported by UK Research and Innovation, and EPSRC Hubs for Robotics and AI in Hazardous Environments: EP/R026092 (FAIR-SPACE), EP/R026173 (ORCA), and EP/R026084 (RAIN).

[1]`http://www.ros.org/`

[2]`https://www.openrobots.org/wiki/genom`

components and, via our calculus for chaining individual specifications, the entire modular system. Once the FOL contracts have been checked for consistency, they can be used to guide the verification of each component. This involves ensuring that the verification encodes the assumptions and guarantees as, for example, test cases, assertions, or formal properties.

Our approach can be used in a Top-Down manner, to guide the system's development from abstract specification to concrete implementation, via verification; or in a Bottom-Up fashion, to check the consistency of existing verification techniques. In this paper we present a Top-Down example. Our approach enables developers to choose the most suitable verification technique for each component, but also to link the conditions being verified across the whole system.

The next section presents background and related work. Section 3 describes our contribution for specifying modular systems using contracts written in FOL; then, we illustrate our approach via a simple remote inspection example. In Section 4 we discuss how the use of distinct forms of verification (e.g., testing, simulation, formal methods, etc.) affects our confidence in the verification results for the whole system. Finally, Section 5 concludes and outlines future directions.

2 Background and Related Work

Modular robotic systems that are used in safety- or mission-critical scenarios require robust verification techniques to ensure and certify that they behave as intended. These techniques encompass a range of software engineering tools and methodologies: from practical testing and simulation, through to formal verification [20, 14]. For example, to show that a property always holds: model-checkers [9] exhaustively explore the search space, and theorem provers [3] use mathematical proof. These techniques offer a means for proving that the software system is correct, which can be used as evidence to improve public trust or to gain certification, as needed. Formal verification can be applied to the implemented system; or to an abstract specification, which can then be further refined to program code.

Many formal methods exist, and most follow a simple paradigm: if the program is executed in a state satisfying a given property (the pre-condition), then it will terminate in a state that satisfies another property (the post-condition) [13]. It is up to the software engineer to specify these properties (sometimes referred to as assumptions and guarantees, respectively). The *Assume-Guarantee* (or Rely-Guarantee) approach [19] enables components to be assessed (verified) individually and then the separate assessments to be combined, potentially under some assumption about the underlying concurrency. To combine assessments, specifications of a component's pre- and post-conditions are needed; this is similar to the notion of Hoare logic that underlies these techniques [17]. In this work, we refer to this combination of assumption/pre-condition and guarantee/post-condition as a *contract* that must be preserved.

Our work takes inspiration from Broy's logical approach to systems engineering [5, 6] in which he distinguishes three kinds of artefacts: (1) system-level requirements, (2) functional system specification, and (3) logical subsystem architecture. Each of these artefacts is represented as a logical predicate in the form of assertions, with relationships defined between them which he then extends to assume/commitment contracts. His treatment of these contracts is purely logical [6], and, in this paper, we present a similar technique that is specialised to the software engineering of modular robotic systems.

In [21], a workflow is proposed for systematically verifying the design of models of a cyber-physical system using a combination of formal refinement and model-checking. Our work also deals with several different levels of abstraction, but we tackle the use of compositional verification in modular systems.

Recent work has analysed a portion of the literature and identified common patterns that appear in robotic missions (e.g. patrolling and obstacle avoidance) [22]. They provide verified LTL/CTL specifica-

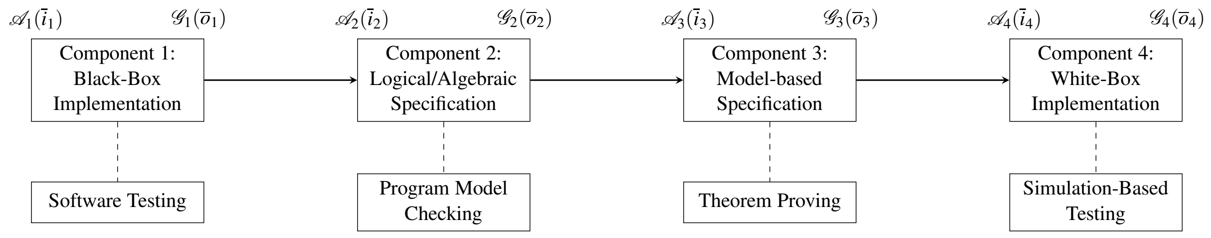

$\mathscr{A}_1(\bar{i}_1)$
$\mathscr{G}_1(\bar{o}_1)$
Component 1: Black-Box Implementation
$\mathscr{A}_2(\bar{i}_2)$
$\mathscr{G}_2(\bar{o}_2)$
Component 2: Logical/Algebraic Specification
$\mathscr{A}_3(\bar{i}_3)$
$\mathscr{G}_3(\bar{o}_3)$
Component 3: Model-based Specification
$\mathscr{A}_4(\bar{i}_4)$
$\mathscr{G}_4(\bar{o}_4)$
Component 4: White-Box Implementation
Software Testing
Program Model Checking
Theorem Proving
Simulation-Based Testing

Figure 1: We specify the contract for each component. Here we denote the assumption/pre-condition by $\mathscr{A}(\bar{i})$ and the guarantee/post-condition by $\mathscr{G}(\bar{o})$. These contracts are used to guide the verification approach applied to each component, denoted by dashed lines, such as software testing for a black-box implementation (Component 1). The solid arrows represent data flow. Note that we use the bar notation $(\bar{i}, \bar{o})$ to indicate a vector of variables.

tions for these commonly found missions which can then be reused in future developments. Further, it is not clear how their support for compositional verification can be extended to support heterogeneous components such as those in our example. Related work includes the identification of Event-B specification clones in cyber physical system specifications [15].

Other compositional approaches include OCRA [8], AGREE [10], CoCoSpec [7], SIMPAL [29], DRONA [12], and a methodology to decompose a system into assume-guarantee contracts that are then validated through simulation [27], however, none explicitly incorporates heterogeneous techniques.

3 A First-Order Logic Framework

No single verification approach suits every component in a modular robotic system [14]. Components such as hardware interfaces or planners may be amenable to formal verification, whereas, sensors may require software testing or simulation-based testing.

As illustrated in Fig. 1, we could use logical specifications (e.g. temporal logic), model-based specifications (e.g. Event-B [1] or Z [28]), or algebraic specifications (e.g. CSP [18] or CASL[24]) amongst others to specify the components in the system. Each of these formalisms offers its own range of benefits, and each tends to suit the verification of particular types of behaviour. Also, we may only have access to the black-box or white-box implementation of a component.

Our approach facilitates the use of compositional verification techniques for the components in modular robotic systems. We achieve this by specifying high-level contracts in FOL and we employ temporal logic for reasoning about the combination of these contracts. In this way, we attach the assumptions over the input ($\mathscr{A}(\bar{i})$) and guarantees over the output ($\mathscr{G}(\bar{o})$) to individual components (see Fig. 1).

3.1 A Calculus for Chaining Component Specifications

Specifically, we use *typed* FOL, potentially with the addition of algebraic operators, to specify assumptions and guarantees. For each component, C, we specify $\mathscr{A}_C(\bar{i}_C)$ and $\mathscr{G}_C(\bar{o}_C)$ where $\bar{i}_C$ is a variable representing the input to the component, $\bar{o}_C$ is a variable representing the output from the component, and $\mathscr{A}_C(\bar{i}_C)$ and $\mathscr{G}_C(\bar{o}_C)$ are FOL formulae describing the assumptions and guarantees, respectively.

Each individual component, C, obeys the following implication:

$$\forall \bar{i}_C, \bar{o}_C \cdot \mathscr{A}_C(\bar{i}_C)[C] \Rightarrow \Diamond \mathscr{G}_C(\bar{o}_C)$$

where $\bar{i}_C$ and $\bar{o}_C$ are the inputs and outputs, respectively, of C; $\mathscr{A}_C(\bar{i}_C)[C]$ represents the execution of C with the assumption $\mathscr{A}_C(\bar{i}_C)$; and '$\Diamond$' is Linear-time Temporal Logic (LTL)'s [25] "eventually" operator.

So, this implication means that if the assumptions, $\mathscr{A}_C(\bar{i}_C)$, hold in the specification or program code of C, then upon execution of the component *eventually* the guarantee, $\mathscr{G}_C(\overline{o}_C)$, will hold. Note that our use of temporal operators here is motivated by the real-time nature of robotic systems and could be of use in later extensions of this calculus for larger, more complex systems.

Components in a modular robotic architecture are 'chained' together so long as their types/requirements match. Similarly, we can compose the contracts of individual components in a number of ways, the simplest being a sequential composition. The basic way to describe such structures is to first have the specification capture *all* of the input and output streams and then to describe how these are combined in the appropriate inference rules. The proof rule, PR1, for such linkage is as expected:

$$\frac{\begin{array}{l}\forall \bar{i}_1, \overline{o}_1.\ \mathscr{A}_1(\bar{i}_1) \Rightarrow \Diamond \mathscr{G}_1(\overline{o}_1)\\ \forall \bar{i}_2, \overline{o}_2.\ \mathscr{A}_2(\bar{i}_2) \Rightarrow \Diamond \mathscr{G}_2(\overline{o}_2)\\ \overline{o_1} = \bar{i}_2\\ \vdash \forall \bar{i}_2, \overline{o}_1.\ \mathscr{G}_1(\overline{o}_1) \Rightarrow \mathscr{A}_2(\bar{i}_2)\end{array}}{\forall \bar{i}_1, \overline{o}_2.\ \mathscr{A}_1(\bar{i}_1) \Rightarrow \Diamond \mathscr{G}_2(\overline{o}_2)} \qquad \text{(PR1)}$$

Intuitively, this states that if two components are sequentially composed with the output of the first equal to the input of the second and the guarantee of the first implies the assumption of the second then, we can deduce that the assumption of the first component implies that the guarantee of the second will "eventually" hold.

The illustrative example that we present in the next section contains a neat, linear chain of components and so it is easy to see how this proof rule works. But a realistic robotic system could be much more complex than this. For example, a component might have multiple, branching output streams that are each used as input to other distinct components. Thus, we propose PR2 to account for branching:

$$\frac{\begin{array}{l}\forall \bar{i}_1, \overline{o}_1.\ \mathscr{A}_1(\bar{i}_1) \Rightarrow \Diamond \mathscr{G}_1(\overline{o}_1)\\ \forall \bar{i}_2, \overline{o}_2.\ \mathscr{A}_2(\bar{i}_2) \Rightarrow \Diamond \mathscr{G}_2(\overline{o}_2)\\ \bar{i}_2 \subseteq \overline{o}_1\\ \vdash \forall \bar{i}_2, \overline{o}_1.\ \mathscr{G}_1(\overline{o}_1) \Rightarrow \mathscr{A}_2(\bar{i}_2)\end{array}}{\forall \bar{i}_1, \overline{o}_2.\ \mathscr{A}_1(\bar{i}_1) \Rightarrow \Diamond \mathscr{G}_2(\overline{o}_2)} \qquad \text{(PR2)}$$

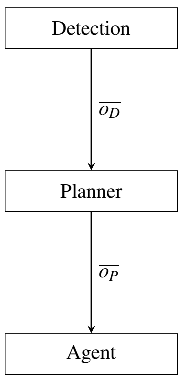
Here, the input to the second component, $\bar{i}_2$, is a subset of the output, $\overline{o}_1$, of the first. We have also devised other proof rules which are omitted here for brevity.

3.2 Remote Inspection Case Study

The various middleware often used to develop robotic systems enable the combination of subsystems, potentially developed independently of one another. For example, an autonomous rover might have a navigation subsystem that is responsible for traversing a planet's surface, and a subsystem for sampling and analysing rock and soil samples.

In this paper, we use the illustrative example of a simplified navigation subsystem for an autonomous rover, whose goal is to perform remote inspection of particular targets, while avoiding any obstacles, on a 2D grid map that has been previously generated. This navigation subsystem is composed of components for *Detection*, a *Planner* and a cognitive *Agent*. Fig. 2 illustrates this subsystem and outlines the abstract FOL contract of each component to be verified.

Given the camera input and the size of the square grid to be explored, n, the specification of the *Detection* component guarantees that it detects an obstacle at a particular coordinate if an obstacle actually exists (in the physical or simulated environment) at that point. Note that we assume the existence



$\overline{i_D}$: camera input, size of square grid to be explored (n)
$\overline{o_D} : Grid = \{(x,y)\}, Obstacles = \{(x,y)\}, s_0 = (x,y)$
$\mathscr{A}_D(\overline{i_D}) : n \in \mathbb{N}$
$\mathscr{G}_D(\overline{o_D}) : \forall x,y \cdot (x,y) \in Obstacles \Rightarrow obstacle(x,y)$
$\quad \wedge\, Grid \subseteq \mathbb{N} \times \mathbb{N} \wedge Obstacles \subseteq Grid \wedge s_0 \in Grid$
$\quad \wedge\, s_0 \notin Obstacles \wedge \forall x,y \cdot (x,y) \in Grid \Rightarrow x < n \wedge y < n$

$\overline{i_P}$: $\overline{o_D}$
$\overline{o_P}$: $PlanSet = \{\{(x,y)\}\}$
$\mathscr{A}_P(\overline{i_P}) : \mathscr{G}_D(\overline{o_D})$
$\mathscr{G}_P(\overline{o_P}) : \forall p \cdot p \in PlanSet \Rightarrow p \subseteq Grid \setminus Obstacles \wedge s_0 \in p$
$\quad \wedge\, \exists p_0 \cdot p_0 \in p \wedge adjacent(s_0, p_0)$
$\quad \forall p_1 \cdot p_1 \in p \wedge p_1 \neq s0 \Rightarrow$
$\quad (\exists r,s \cdot r \in p \wedge s \in p \wedge adjacent(r,p_1) \wedge adjacent(p_1,s))$

$\overline{i_A}$: $\overline{o_P}$
$\overline{o_A}$: $plan = \{(x,y)\}$
$\mathscr{A}_A(\overline{i_A}) : \mathscr{G}_P(\overline{o_P})$
$\mathscr{G}_A(\overline{o_A}) : plan \in PlanSet \wedge \forall q \cdot q \in PlanSet \Rightarrow | plan | \leq | q |$

```
1  Event Planner ≙ordinary
2   any p
3   where
4     grd1: goal ≠ s₀  ∧ goal ∈ p
5     grd2: p ⊆ Grid \ Obstacles  ∧  s₀ ∈ p
6     grd4: ∃p₀ · p₀ ∈ p ∧ adjacent(s₀ ↦ p₀)
7     grd6: ∀p₁ · p₁ ∈ p ∧ p₁ ≠ s₀ ∧ p₁ ≠ goal
   ⇒ (∃r, s · r ∈ p ∧ s ∈ p ∧  adjacent(r ↦ p₁) ∧
   adjacent(p₁ ↦ s))
8     grd5: ∃p₂ · p₂ ∈ p ∧ adjacent(p₂ ↦ goal)
9   then
10    act1: PlanSet  :=  PlanSet  ∪  {p}
```

Figure 2: An overview of the robotic system to be verified. On the far left, each rectangle represents a component, and each arrow represents data flow between components. To the right of each individual component, we summarise the $\bar{i}$s, $\bar{o}$s, and their respective FOL contract. On the far right, we provide an Event-B event specification corresponding to the *Planner*. Here, '$\mapsto$' denotes tuples.

of the $obstacle(x,y)$ function to represent this physical or simulated observation. Further, the *Detection* component outputs the rover's current location, denoted by s_0, and guarantees that s_0 is in the grid and does not coincide with an obstacle.

The *Planner* component produces a *PlanSet* whose elements are sets of coordinates capturing obstacle-free plans under the assumption that the *Detection* component has produced a set of *Obstacles* that refers to points that exist and contain an obstacle. The guarantee also ensures that each of these plans (sets of coordinates) in *PlanSet* can be transformed into a sequence of coordinates where each is adjacent to the previous. Note that we use sets here rather than sequences because sequences may contain duplicate entries whereas sets cannot, and so our specification rules out the case where a plan loops.

The specification of the cognitive *Agent* component states that, under the assumption that all potential plans in the *PlanSet* are obstacle-free, then it chooses a *plan* from the *PlanSet* and that this plan is, in fact, the shortest one available. Note that we use $| plan |$ to denote the length of *plan*.

The far right of Fig. 2 contains an Event-B event specification [1] of the *Planner* component. Notice that the high-level FOL contract that we have written is captured in the Event-B specification as guards on lines 5–7. The FOL contract has been refined and the Event-B specification has a notion of a goal that it must plan paths towards. This has resulted in the extra guards on lines 1 and 8. This is an excerpt from a larger *Planner* specification that we have verified via theorem proving in the *Rodin* Platform [2].

In this illustrative example, we formally verified the *Planner* contracts using Event-B and the *Agent* contracts using the Gwendolen agent programming language [11]. The *Detection* component would be verified via standard software testing practices and simulation-based testing since it may contain a learning component which cannot be formally verified.

In §3.1, we presented rules about how we chain contracts together. In this example, two applications of PR1 would allow us to derive the following property:

$$\forall \bar{i}_D, \bar{o}_A \cdot \mathscr{A}_C(\bar{i}_D) \Rightarrow \Diamond \mathscr{G}_A(\bar{o}_A)$$

meaning that if the *Detection* component receives valid input then the *Agent* will eventually return the shortest path from the start to the goal position in the grid.

We have shown how FOL can be used as a unifying language for the high-level Assume-Guarantee specification of components in a modular robotic system. A top-down approach to software engineering of such a robotic system would begin with these FOL contracts. Then, each contract would be further refined [23], ideally to a more detailed formal specification and, eventually to its concrete implementation.

4 Confidence in Verification

When linking the results from multiple verification techniques a key question becomes how using these different techniques affects our confidence in the verification of the whole system. One might think that a formal proof of correctness corresponds to a higher level of confidence than simple testing methods (especially over unbounded environments). However, formal verification is usually only feasible on an abstraction of the system whereas testing can be carried out on the implemented code. Therefore, it is our view that we achieve higher levels of confidence in verification when multiple verification methods have been employed for each component in the system [30].

Component	**Testing**	**Simulation-Based Testing**	**Formal Methods**
Detection	✓	✗	✗
Planner	✓	✓	✓
Cognitive Agent	✓	✓	✓

Table 1: Verification techniques applied to each component.

We have broadly partitioned current verification techniques into three categories: testing, simulation-based testing and formal methods. We have determined which of these techniques might be employed for each component in our example as shown in Table 1. Examining how this metric can be calculated for more complex systems with loops is a future direction for this work.

5 Conclusions and Future Work

We have presented an initial description of a framework for compositional verification of modular systems using FOL as a unifying language. FOL contracts are used to guide the verification techniques applied to each component. We have briefly illustrated this by verifying and refining the FOL contract for the *Planner* component using Event-B in our example of a remote inspection rover. Furthermore, we introduce the notion of confidence in verification techniques and provide a broad categorisation.

Future work includes assessing how well our approach scales to industrial-sized robotic systems development. Also, ensuring that our approach is usable by developers and understandable by certification organisations is crucial to its use and success. This needs tool support for writing and reasoning about the FOL contracts, with similar functionality to an IDE, integrated robotic development other tools. We will also demonstrate the approach with a wider variety of verification techniques. Our illustrative example employs an event-based style of communication, however, we intend to explore how to extend our approach to stream-based communications as future work.

Finally, our proposed method could be used to generate runtime monitors to check that each component's contract holds during execution. If a contract is violated, the monitor could flag this to the robot's operator (if one exists) or trigger a mitigating action (if the robot is autonomous). This could augment the verification of components that can only be verified at a low level of confidence, or aid in fault finding when diagnosing failures.

References

[1] Jean-Raymond Abrial (2010): *Modeling in Event-B*. Cambridge University Press, doi:10.1017/CBO9781139195881.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta & Laurent Voisin (2010): *Rodin: an open toolset for modelling and reasoning in Event-B*. International Journal on Software Tools for Technology Transfer 12(6), pp. 447–466, doi:10.1007/s10009-010-0145-y.

[3] Yves Bertot & Pierre Castéran (2013): *Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions*. Springer, doi:10.1007/978-3-662-07964-5.

[4] Robert Bogue (2011): *Robots in the nuclear industry: a review of technologies and applications*. Industrial Robot: An International Journal 38(2), pp. 113–118, doi:10.1108/01439911111106327.

[5] Manfred Broy (2018): *A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability – from requirements to functional and architectural views*. Software and System Modeling 17(2), pp. 365–393, doi:10.1007/s10270-017-0619-4.

[6] Manfred Broy (2018): *Theory and methodology of assumption/commitment based system interface specification and architectural contracts*. Formal Methods in System Design 52(1), pp. 33–87, doi:10.1007/s10703-017-0304-9.

[7] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai & Cesare Tinelli (2016): *CoCoSpec: A mode-aware contract language for reactive systems*. In: International Conference on Software Engineering and Formal Methods, LNCS 9763, Springer, pp. 347–366, doi:10.1007/978-3-319-41591-8_24.

[8] Alessandro Cimatti, Michele Dorigatti & Stefano Tonetta (2013): *OCRA: A tool for checking the refinement of temporal contracts*. In: International Conference on Automated Software Engineering (ASE), IEEE, pp. 702–705, doi:10.1109/ASE.2013.6693137.

[9] Edmund M Clarke, Orna Grumberg & Doron Peled (1999): *Model checking*. MIT press.

[10] Darren Cofer, Andrew Gacek, Steven Miller, Michael W Whalen, Brian LaValley & Lui Sha (2012): *Compositional verification of architectural models*. In: NASA Formal Methods Symposium, LNCS 7226, Springer, pp. 126–140, doi:10.1007/978-3-642-13464-7_5.

[11] Louise A. Dennis, Michael Fisher, Matthew P. Webster & Rafael H. Bordini (2012): *Model checking agent programming languages*. Automated Software Engineering 19(1), pp. 5–63, doi:10.1007/s10515-011-0088-x.

[12] Ankush Desai, Shaz Qadeer & Sanjit A. Seshia (2018): *Programming Safe Robotics Systems: Challenges and Advances*. In Tiziana Margaria & Bernhard Steffen, editors: Leveraging Applications of Formal Methods, Verification and Validation. Verification, Springer International Publishing, Cham, pp. 103–119, doi:10.1007/978-3-030-03421-4_8.

[13] Edsger W Dijkstra (1975): *Guarded commands, nondeterminacy and formal derivation of programs*. Communications of the ACM 18(8), pp. 453–457, doi:10.1145/360933.360975.

[14] Marie Farrell, Matt Luckcuck & Michael Fisher (2018): *Robotics and Integrated Formal Methods: Necessity meets Opportunity*. In: Integrated Formal Methods, LNCS 11023, Springer, pp. 161–171, doi:10.1007/978-3-319-98938-9_10.

[15] Marie Farrell, Rosemary Monahan & James F Power (2017): *Specification Clones: An Empirical Study of the Structure of Event-B Specifications*. In: International Conference on Software Engineering and Formal Methods, LNCS 10469, Springer, pp. 152–167, doi:10.1007/978-3-319-66197-1_10.

[16] Helen F. Hastie, Katrin Solveig Lohan, Mike J. Chantler, David A. Robb, Subramanian Ramamoorthy, Ronald P. A. Petrick, Sethu Vijayakumar & David Lane (2018): *The ORCA Hub: Explainable Offshore Robotics through Intelligent Interfaces*. CoRR abs/1803.02100. Available at http://arxiv.org/abs/1803.02100.

[17] C. A. R. Hoare (1969): *An axiomatic basis for computer programming*. Communications of the ACM 12(10), pp. 576–580, doi:10.1145/363235.363259.

[18] C. A. R. Hoare (1978): *Communicating sequential processes*. Communications of the ACM 21(8), pp. 666–677, doi:10.1145/359576.359585.

[19] Cliff B. Jones (1983): *Tentative Steps Toward a Development Method for Interfering Programs*. ACM Transactions on Programming Languages and Systems 5(4), pp. 596–619, doi:10.1145/69575.69577.

[20] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon & Michael Fisher (2019): *Formal Specification and Verification of Autonomous Robotic Systems: A Survey*. ACM Comput. Surv. 52(5), pp. 1–41, doi:10.1145/3342355.

[21] Christoph Luckeneder & Hermann Kaindl (2018): *Systematic top-down design of cyber-physical models with integrated validation and formal verification*. In: International Conference on Software Engineering, pp. 274–275, doi:10.1145/3183440.3194967.

[22] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi & Thorsten Berger (2019): *Specification patterns for robotic missions*. IEEE Transactions on Software Engineering, doi:10.1109/TSE.2019.2945329.

[23] Carroll Morgan, Ken Robinson & Paul Gardiner (1988): *On the Refinement Calculus*. Springer, doi:10.1007/978-1-4471-3273-8.

[24] Peter D Mosses (2004): *CASL reference manual: The complete documentation of the common algebraic specification language*. Springer, doi:10.1007/b96103.

[25] A. Pnueli (1977): *The Temporal Logic of Programs*. In: 18th Symposium on the Foundations of Computer Science, IEEE, pp. 46–57, doi:10.1109/SFCS.1977.32.

[26] Azamat Shakhimardanov, Nico Hochgeschwender & Gerhard Kraetzschmar (2010): *Component models in robotics software*. In: Workshop on Performance Metrics for Intelligent Systems, ACM, pp. 82–87, doi:10.1145/2377576.2377592.

[27] Stefano Spellini, Michele Lora, Franco Fummi & Sudipta Chattopadhyay (2019): *Compositional Design of Multi-Robot Systems Control Software on ROS*. ACM Trans. Embed. Comput. Syst. 18(5s), doi:10.1145/3358197.

[28] J Michael Spivey (1988): *Understanding Z: a specification language and its formal semantics*. Cambridge University Press.

[29] Lucas Wagner, David Greve & Andrew Gacek (2017): *SIMPAL: A Compositional Reasoning Framework for Imperative Programs*. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, SPIN 2017, Association for Computing Machinery, New York, NY, USA, p. 9093, doi:10.1145/3092282.3092290.

[30] Matt Webster, David Western, Dejanira Araiza-Illan, Clare Dixon, Kerstin Eder, Michael Fisher & Anthony G Pipe (2020): *A corroborative approach to verification and validation of humanrobot teams*. The International Journal of Robotics Research 39(1), pp. 73–99, doi:10.1177/0278364919883338.

[31] Brian H. Wilcox (1992): *Robotic vehicles for planetary exploration*. Applied Intelligence 2(2), pp. 181–193, doi:10.1007/BF00058762.

From Requirements to Autonomous Flight: An Overview of the Monitoring ICAROUS Project

Aaron Dutle[1] Laura Titolo[2] Dimitra Giannakopoulou[3]
César Muñoz[1] Ivan Perez[2] Anastasia Mavridou[4]
Esther Conrad[1] Swee Balachandran[2] Thomas Pressburger[3]
Alwyn Goodloe[1]

[1]NASA Langley Research Center, Hampton, Virginia
[2]National Institute of Aerospace, Hampton, Virginia
[3]NASA Ames Research Center, Moffett Field, California
[4]KBR Inc. / NASA Ames Research Center, Moffett Field, California

{*aaron.m.dutle, cesar.a.munoz, esther.d.conrad, a.goodloe, laura.titolo, ivan.perezdominguez, sweewarman.balachandran, dimitra.giannakopoulou, anastasia.mavridou, tom.pressburger*}*@nasa.gov*

The Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS) is a software architecture incorporating a set of algorithms to enable autonomous operations of unmanned aircraft applications. This paper provides an overview of *Monitoring ICAROUS*, a project whose objective is to provide a formal approach to generating runtime monitors for autonomous systems from requirements written in a structured natural language. This approach integrates FRET, a formal requirement elicitation and authoring tool, and Copilot, a runtime verification framework. FRET is used to specify formal requirements in structured natural language. These requirements are translated into temporal logic formulae. Copilot is then used to generate executable runtime monitors from these temporal logic specifications. The generated monitors are directly integrated into ICAROUS to perform runtime verification during flight.

1 Introduction

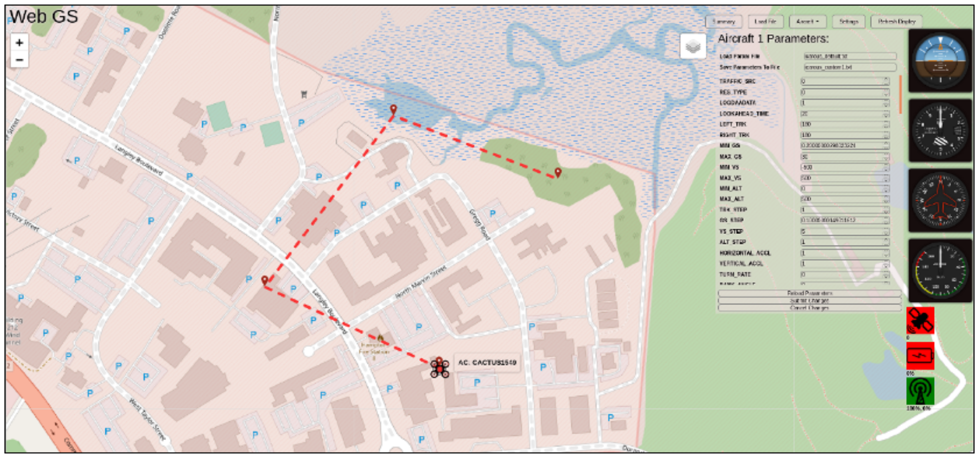
The Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS) [5] is a software architecture for enabling safe autonomous operation of unmanned aircraft systems (UAS) in the airspace. The primary goal of ICAROUS is to provide autonomy to enable beyond visual line of sight (BVLOS) missions for UAS without the need for constant human supervision/intervention. ICAROUS provides highly-assured functions to avoid stationary obstacles, maintain a safe distance from other users of the airspace, and compute resolution and recovery maneuvers.

Hardware and software verification via formal methods offers the highest assurance of safety available for such cyber-physical systems. While there have been considerable advances in creating industrial-scale formal methods (e.g., [6, 12, 23]), it is not yet practical to apply them to an entire complex system such as ICAROUS. Formal verification is generally carried out on a model of a system rather than the software itself, and so the properties verified may not hold if the model is inaccurate or if other faults make the system behave unpredictably. Moreover, while there has been much progress made in verification of neural networks in particular ([11, 13], increasingly autonomous systems employing machine learning and similar methods are challenging for formal verification.

Runtime verification (RV) [10] is a verification technique that has the potential to enable the safe operation of safety-critical systems that are too complex to formally verify or fully test. In RV, the system is monitored during execution, and property violations can be detected and acted upon during the mission. RV detects when properties are violated at runtime, so it cannot enforce the correct operation of a system, but is an improvement over testing alone, and can enhance testing by finding real cases of requirement violation. Copilot [17] is a runtime verification framework developed by NASA researchers and others.

While RV can be used to monitor and detect property violations, the actual properties to be monitored must be determined and specified externally. Such safety requirements are generally written by hand in natural language, which can lead to ambiguity as to their meaning or applicability. Additionally, when runtime verification is used as a key safety component of an autonomous system, having clearly specified requirements that are properly translated into *executable* monitors is critical. FRETISH [9] is a structured natural language developed by NASA to write unambiguous requirements. The associated tool, FRET (Formal Requirements Elicitation Tool), provides a framework to write, formalize and analyze requirements and automatically generate temporal logic formulae from them.

The *Monitoring ICAROUS* project, a work-in-progress joint effort at NASA, will demonstrate the integration of robust requirements-based runtime verification applied to an autonomous flight system for unmanned aircraft using FRET, Copilot, and ICAROUS. This project brings together work that the NASA formal methods team has been doing for many years on requirements elicitation and specification, runtime verification, and assured autonomous aircraft software.



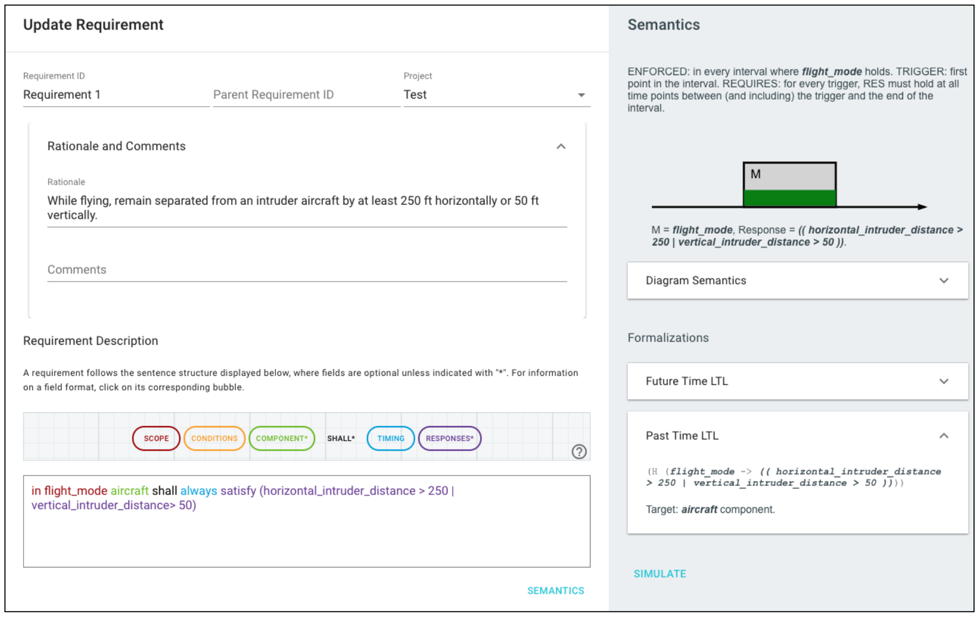
Aircraft 1 Parameters:

Load Param File	[illegible]
Save Parameters To File	icarous_custom1.txt
TRAFFIC_SRC	0
RES_TYPE	0
LOGDAADATA	1
LOOKAHEAD_TIME	20
LEFT_TRK	180
RIGHT_TRK	180
MIN_GS	[illegible]
MAX_GS	50
MIN_VS	-500
MAX_VS	500
MIN_ALT	0
MAX_ALT	500
TRK_STEP	1
GS_STEP	[illegible]
VS_STEP	5
ALT_STEP	1
HORIZONTAL_ACCL	1
VERTICAL_ACCL	1
TURN_RATE	0

Figure 1: The current interface for setting parameters in ICAROUS.

The concept of operation for the integrated system is simple. Prior to the start of an autonomous flight with ICAROUS, an operator can set a collection of different mission and safety parameters (see Figure 1). For example, the detect-and-avoid module can be specified to avoid other aircraft by at least 250 ft horizontally and 50 ft vertically. With integrated monitoring, the related implicit requirements will become explicit ones, expressed in the structured natural language of FRETISH, and available to be viewed and edited by the user. In addition, a method for specifying custom requirements similar to the FRET interface will be available. These requirements expressed in FRETISH will be translated into Copilot's monitoring language, which will generate C code for the RV monitors. These monitors will be integrated into ICAROUS, which will use them to determine requirements violations. A violation of a monitor will alert the operator, who can use the information to return the aircraft to a safe state.

As a motivating example, the remainder of the paper will use the detect and avoid requirement *"Requirement 1: While flying, remain separated from an intruder aircraft by at least 250 ft horizontally or 50 ft vertically."* This safety property will be followed through the chain of tools employed, and illustrate



Update Requirement

Requirement ID: Requirement 1
Parent Requirement ID
Project: Test

Rationale and Comments

Rationale: While flying, remain separated from an intruder aircraft by at least 250 ft horizontally or 50 ft vertically.

Comments

Requirement Description

A requirement follows the sentence structure displayed below, where fields are optional unless indicated with "*". For information on a field format, click on its corresponding bubble.

in flight_mode aircraft shall always satisfy (horizontal_intruder_distance > 250 | vertical_intruder_distance> 50)

Semantics

ENFORCED: in every interval where ***flight_mode*** holds. TRIGGER: first point in the interval. REQUIRES: for every trigger, RES must hold at all time points between (and including) the trigger and the end of the interval.

M = ***flight_mode***, Response = ***((horizontal_intruder_distance > 250 | vertical_intruder_distance > 50))***.

```
(H (flight_mode -> (( horizontal_intruder_distance
> 250 | vertical_intruder_distance > 50 ))))
```

Target: ***aircraft*** component.

Figure 2: FRET editor, with the example requirement entered, and the *Semantics* pane visible.

the work to be done in the integration.

2 Tool Descriptions

FRET[1] is an open-source tool developed at NASA for writing, understanding, formalizing, and analyzing requirements. 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 textual and diagrammatic explanations to clarify subtle semantic issues.

Figure 2 illustrates FRET's requirements elicitation interface, with the example *Requirement 1* entered. The "Rationale and Comments" field holds the original text requirement; the "Requirement Description" field is where the FRETISH requirement is composed. Once a requirement is entered, the "Semantics" pane shows a text description of the FRETISH requirement, displays a "semantic diagram" showing a visual explanation of the requirement applicability over time, and provides translations from FRETISH to Metric Future- and Past-time linear temporal logic (LTL) [14].

A FRETISH requirement description is automatically parsed into six sequential fields; *scope, condition, component, shall, timing,* and *response*, with the FRET editor dynamically coloring the text corresponding to each field (Fig. 2). The mandatory *component* field specifies the component that the requirement applies to (**aircraft**). 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 (**horizontal_intruder_distance**$>$**250** $|$ **vertical_intruder_distance**$>$**50**). The (optional) *scope* field states that the requirement is only assessed during particular modes, for the example, in **flight_mode**. The (optional) Boolean expression field *condition* states that, within the specified mode,

[1] `https://github.com/NASA-SW-VnV/fret`

the requirement becomes relevant only from the point where the condition becomes true. The (optional) *timing* field specifies at which points the response must occur, in the example “always”, meaning at all points in flight mode.

FRET automatically produces formulas in several formal languages, including Metric Future-time LTL and Past-time LTL. FRET also offers an interactive visualizer, showing temporal traces of each of the signals (variables) involved as well as the valuation of the requirement for each point in time.

Copilot[2] is an open-source runtime verification framework for real-time embedded systems. Copilot monitors are written in a compositional, stream-based language. The framework translates monitor specifications into C99 code with no dynamic memory allocation and executes with predictable memory and time, crucial in resource-constrained environments, embedded systems, and safety-critical systems.

The Copilot language has been designed to be high-level, easy to understand, and robust. To prevent errors in the RV system that could affect systems during a mission, the language uses advanced programming features to provide additional compile-time and runtime guarantees. For example, all arrays in Copilot have fixed length, which makes it possible for the system to detect, before the mission, some array accesses that would be out of bounds.

Copilot supports a number of logical formalisms for writing specifications including a bounded version of Future-time Linear Temporal Logic [20], Past-time Linear Temporal Logic (PTLTL) [15], and Metric Temporal Logic (MTL) [14]. Copilot also includes support libraries with functions such as majority vote, used to implement fault-tolerant monitors [19]. *Requirement 1* expressed as a PTLTL Copilot specification is as follows:

```
alwaysBeen (flightMode ==> (  horizontalIntruderDistance > 250
                           || verticalIntruderDistance   > 50 ))
```

ICAROUS[3] is an open-source software architecture developed at NASA to enable safety-centric autonomous aircraft missions. It is a service-oriented architecture, where service applications provide various capabilities such as path planning, sense and avoid, geofence containment, task planning, and more, through a publish-subscribe middleware. Applications are logically organized into conflict detectors, conflict resolvers, mission managers, and decision makers.

Conflict detectors are algorithms that check for imminent violation of constraints such as geofences, conflicts due to other vehicles in the airspace, deviations from mission flight plan, etc. These conflict detecting applications can also provide *tactical* resolutions, which provide a simple maneuver which will prevent the corresponding conflict. Conflict resolvers compute resolutions to prevent imminent violation of specified constraints. Resolvers may handle multiple conflicts simultaneously, and can provide *strategic* resolutions that are computed to prevent one or more constraint violations. A decision making application receives conflict information from monitors and triggers resolvers to compute resolutions for one or more conflicts. When resolving imminent constraint violation, outputs from mission applications are ignored. The mission is resumed once all conflicts are resolved.

These services are connected through the NASA core Flight System (cFS) middleware,[4] a platform-independent reusable software framework and a set of reusable software applications. The three key aspects to the cFS architecture are a dynamic run-time environment, layered software, and a component-based design. These key aspects make cFS suitable for reuse on any number of embedded software systems. The cFS middleware simplifies the flight software development process by providing the un-

[2]`https://copilot-language.github.io/`

[3]`https://github.com/nasa/icarous`

[4]`https://cfs.gsfc.nasa.gov/`

derlying infrastructure and hosting a runtime environment for development of project/mission specific applications.

3 Integration

The integration of the three systems described in Section 2 requires several steps in order to create a complete framework. The three major steps in the integration are as follows. The first step is developing a method for ICAROUS-specific requirements and safety properties to be specified in FRET. Next, the requirements expressed in FRETISH must be translated into Copilot. Finally, the monitors generated by Copilot must be integrated into ICAROUS in a usable way. Each of these integration steps are discussed in turn below, and the toolchain is depicted in Figure 3.

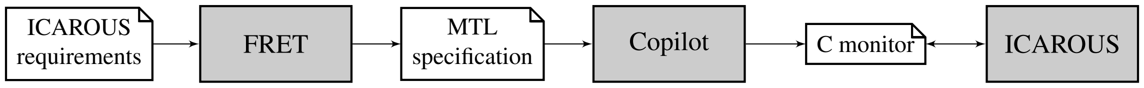


Figure 3: Toolchain to automatically generate monitors for ICAROUS.

3.1 ICAROUS interface to FRET

In order to facilitate the use of FRET for ICAROUS-specific requirements, several things need to be done. The first of these is a thorough accounting of all of the modes, systems, and signals available within ICAROUS for a monitor to access.

In FRET, the specification of safety properties is completely independent of the system that is being considered. This makes FRET very general and powerful in its ability to craft requirements, but makes it somewhat cumbersome to use in the specific setting of ICAROUS. The variables that FRET uses to refer to modes, systems, and signals in requirements are completely arbitrary, while being able to use such a requirement as a monitor means that each of these variables must correspond to an actual system or signal in ICAROUS. After each of these possible variables in ICAROUS is identified, along with its datatype information, FRET can be restricted to using only information that can be obtained by the system in specifying properties.

Another task to be completed is the generation of ICAROUS-specific templates for FRET. Many of the safety requirements that an autonomous flight system will be expected to follow exhibit similar patterns from flight to flight. For example, an autonomous operation may be required to stay below a certain altitude ceiling. More specific requirements such as *Requirement 1* are parametric requirements that a detect-and-avoid (DAA) system is expected to obey based on settings in the DAA system. For such requirements, a template can be given allowing the user to input particular values based on the mission, and the associated requirement added to those to be monitored. Additionally, many of the existing settings in an ICAROUS configuration carry with them implicit safety requirements. Setting the *max_altitude* parameter in the DAA system of ICAROUS can be interpreted as a requirement that the aircraft never goes above the value set. Parsing these files can allow for the *automatic* generation of a requirement from the associated template and the variable setting.

3.2 FRET to Copilot translation

The main integration step between FRET and Copilot is to take a requirement expressed in FRETISH, and translate it into a Copilot monitor. A prototype tool named Ogma is being developed to create Copilot monitors from languages such as FRETISH, SPEAR [7], and AGREE [4]. The tool can translate the Past-time LTL formulas FRET generates, so it can process any requirements specified in FRETISH. The resulting Copilot monitor can then be automatically translated into C99 code that checks that a corresponding property holds during runtime. This conversion should occur transparently: users specify FRETISH requirements, and automatically obtain C code compatible with ICAROUS.

3.3 Copilot Monitors in ICAROUS

The integration of Copilot-generated monitors into ICAROUS should be fairly straight-forward. Since ICAROUS is already a service-oriented publish/subscribe architecture, the main issue is subscribing and routing the appropriate signals to the monitors, and returning a signal to the user that indicates which properties have been violated. To facilitate this integration, the Ogma tool automatically generates a cFS application, responsible for subscribing to the appropriate ICAROUS applications, making data available to Copilot, and handle runtime violations reported by Copilot.

A technical difficulty in this approach (mentioned in Section 3.1) is that the RV monitors are generated *after* requirements are specified, and then this C code is integrated into ICAROUS. Currently, ICAROUS is installed on the aircraft once, and, generally, only settings are changed between flights. With new monitors generated for each flight, ICAROUS must be configured, new monitors generated, and then the system installed on the aircraft before flight. A partial solution would be to include parametric versions of common monitors, and have the parameters instantiated prior to flight. Alternatively, a utility for including monitoring code could be added, since the RV service would not change.

4 Conclusion

The work described here is still in-progress. Additional work is being conducted tangential to this integration. The FRETISH language and semantics are being specified in the Prototype Verification System (PVS) [16], to prove that the evaluation of a FRETISH statement is the same as the evaluation of the LTL statement produced by FRET for all possible finite and infinite traces. Currently, the verification of the translation is done through a systematic, rigorous testing framework for finite traces up to a certain length. An embedding of FRETISH in PVS would also allow for formal reasoning about models of a system such as ICAROUS with respect to specified requirements.

Related work on requirements specification, RV, and autonomous flight systems is omitted here. The interested reader is directed to the corresponding sections of [8, 9] for requirements, [2, 10, 17, 18] for runtime verification, and [1, 5] for autonomous flight systems. Work that has a similar flavor to the integration of these tools includes [3], where the R2U2 [22] engine is used to monitor an automated and intelligent UAS Traffic Management System for adherence to safety requirements during operation. The specifications are written in the Mission-time Linear Temporal Logic (MLTL) [21], an extension of MTL, in contrast with the present approach where the specifications are given using structured natural language.

The *Monitoring ICAROUS* project will allow a user to obtain relevant requirements automatically or defined using the structured natural language FRETISH, translate these requirements into runtime monitors using Copilot, and seamlessly integrate these monitors into the autonomous flight system ICAROUS.

The framework supports simple requirements specification and analysis, with robust runtime verification, while the translation and integration steps are performed in the background. Requirements-based runtime monitoring demonstrates a real-world application of formal methods to increase the safety assurance of complex automated systems.

References

[1] S. Balachandran, C. Muñoz, M. Consiglio, M. Feliú & A. Patel (2018): *Independent Configurable Architecture for Reliable Operation of Unmanned Systems with Distributed On-Board Services*. In: *Proceedings of the 37th Digital Avionics Systems Conference (DASC 2018)*, pp. 1–6, doi:10.1109/DASC.2018.8569752.

[2] E. Bartocci, Y. Falcone, A. Francalanza & G. Reger (2018): *Introduction to Runtime Verification*. In: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *Lecture Notes in Computer Science* 10457, Springer, pp. 1–33, doi:10.1007/978-3-319-75632-5_1.

[3] M. Cauwels, A. Hammer, B. Hertz, P. Jones & K. Y. Rozier (2020): *Integrating Runtime Verification into an Automated UAS Traffic Management System*. In: *International workshop on moDeling, vErification and Testing of dEpendable CriTical systems, DETECT 2020*, pp. 340–357, doi:10.1007/978-3-030-59155-7_26.

[4] D. D. Cofer, Gacek. A., S. P. Miller, M. W. Whalen, B. LaValley & L. Sha (2012): *Compositional Verification of Architectural Models*. In: *Proceedings of the 4th International NASA Formal Methods Symposium (NFM 2012)*, *Lecture Notes in Computer Science* 7226, Springer, pp. 126–140, doi:10.1007/978-3-642-28891-3_13.

[5] M. Consiglio, C. Muñoz, G. Hagen, A. Narkawicz & S. Balachandran (2016): *ICAROUS: Integrated Configurable Algorithms for Reliable Operations of Unmanned Systems*. In: *Proceedings of the 35th Digital Avionics Systems Conference (DASC 2016)*, pp. 1–5, doi:10.1109/DASC.2016.7778033.

[6] Byron Cook (2018): *Formal Reasoning About the Security of Amazon Web Services*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 38–47, doi:10.1007/978-3-319-96145-3_3.

[7] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello & J. A. Davis (2017): *SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements*. In: *Proceedings of the 9th International NASA Formal Methods Symposium (NFM 2017)*, *Lecture Notes in Computer Science* 10227, pp. 420–426, doi:10.1007/978-3-319-57288-8_30.

[8] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann & N. Shi (2020): *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)*.

[9] D. Giannakopoulou, T. Pressburger, A. Mavridou & J. Schumann (2020): *Generation of Formal Requirements from Structured Natural Language*. In: *26th International Working Conference on Requirements Engineering: Foundation for Software Quality, REFSQ 2020*, *Lecture Notes in Computer Science* 12045, Springer, pp. 19–35, doi:10.1007/978-3-030-44429-7_2.

[10] K. Havelund & A. Goldberg (2008): *Verify Your Runs*, pp. 374–383. *Lecture Notes in Computer Science* 4171, Springer, doi:10.1007/978-3-540-69149-5_40.

[11] K. Julian & M. Kochenderfer (2019): *Guaranteeing Safety for Neural Network-Based Aircraft Collision Avoidance Systems*. *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pp. 1–10, doi:10.1109/DASC43569.2019.9081748.

[12] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber et al. (2009): *Replacing Testing with Formal Verification in Intel® CoreTM i7 Processor Execution Engine Validation*. In: *Computer Aided Verification*, Springer, pp. 414–429, doi:10.1007/978-3-642-02658-4_32.

[13] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer & C. Barrett (2019): *The Marabou Framework for Verification and Analysis of Deep Neural Networks*. In I. Dillig & S. Tasiran, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 443–452, doi:10.1007/978-3-030-25540-4_26.

[14] R. Koymans (1990): *Specifying Real-time Properties with Metric Temporal Logic*. *Real-Time Syst.* 2(4), pp. 255–299, doi:10.1007/BF01995674.

[15] F. Laroussinie, N. Markey & P. Schnoebelen (2002): *Temporal Logic with Forgettable Past*. In: *LICS02: Proceeding of Logic in Computer Science 2002*, IEEE Computer Society Press, pp. 383–392, doi:10.1109/LICS.2002.1029846.

[16] S. Owre, J. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System*. In: *Proceeding of the 11th International Conference on Automated Deduction (CADE)*, *Lecture Notes in Artificial Intelligence* 607, Springer, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[17] I. Perez, F. Dedden & A. Goodloe (2020): *Copilot 3*. Technical Report NASA/TM2020220587, NASA Langley Research Center, doi:10.13140/RG.2.2.35163.80163.

[18] L. Pike, A. Goodloe, R. Morisset & S. Niller (2010): *Copilot: A Hard Real-Time Runtime Monitor*. In: *Proceedings of the First International Conference on Runtime Verification (RV 2010)*, *Lecture Notes in Computer Science* 6418, Springer, pp. 345–359, doi:10.1007/978-3-642-16612-9_26.

[19] L. Pike, N. Wegmann, S. Niller & A. Goodloe (2013): *Copilot: monitoring embedded systems*. *Innovations in Systems and Software Engineering* 9(4), pp. 235–255, doi:10.1007/s11334-013-0223-x.

[20] A. Pnueli (1977): *The Temporal Logic of Programs*. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, IEEE Computer Society, Washington, DC, USA, pp. 46–57, doi:10.1109/SFCS.1977.32.

[21] T. Reinbacher, K. Y. Rozier & J. Schumann (2014): *Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*. In: *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, *Lecture Notes in Computer Science* 8413, Springer, pp. 357–372, doi:10.1007/978-3-642-54862-8_24.

[22] J. Schumann, P. Moosbrugger & K. Y. Rozier (2015): *R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems*. In: *Proceedings of the 6th International Conference on Runtime Verification (RV 2015)*, *Lecture Notes in Computer Science* 9333, Springer, pp. 233–249, doi:10.1007/978-3-319-23820-3_15.

[23] J. Souyris, V. Wiels, D. Delmas & H. Delseny (2009): *Formal Verification of Avionics Software Products*. In: *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, Springer-Verlag, Berlin, Heidelberg, p. 532546, doi:10.1007/978-3-642-05089-3_34.

YAP: Tool Support for Deriving Safety Controllers from Hazard Analysis and Risk Assessments

Mario Gleirscher[*]
Dept. of Computer Science, University of York, York, U.K.
mario.gleirscher@york.ac.uk

Safety controllers are system or software components responsible for handling risk in many machine applications. This tool paper describes a use case and a workflow for YAP, a research tool for risk modelling and discrete-event safety controller design. The goal of this use case is to derive a safety controller from hazard analysis and risk assessment, to define a design space for this controller, and to select a verified optimal controller instance from this design space. We represent this design space as a stochastic model and use YAP for risk modelling and generation of parts of this stochastic model. For the controller verification and selection step, we use a stochastic model checker. The approach is illustrated by an example of a collaborative robot operated in a manufacturing work cell.

1 Introduction

To ensure their safe operation, machines, such as mobile robots or delivery drones, incorporate controllers responsible for the *handling of critical events (CEs)* while performing their tasks. We will refer to such controllers as *safety controllers*. CEs can be failures, human errors, or other hazards, any causes thereof and any consequences, such as incidents or accidents. CE handling can involve the anticipation and mitigation of hazards and the prevention and alleviation of accidents, for example, by switching a machine into a mode with lower risk, that is, a *safety mode*, by performing a *safety function* (e.g. a warning signal), or by changing the machine's activity. Therefore, safety controllers have to be carefully specified, designed, and verified in order to be deployed according to state-of-the-art regulations [11, 9].

This tool paper supplements our approach to the verified synthesis of safety controllers [6] with a hands-on guide to the research tool YAP *Against Perils* [5]. One objective of YAP is to support the steps required to transform results from hazard analysis into verifiable models of safety controllers. YAP seeks to bridge the gap between the identification of hazards, the formulation of safety goals, and the implementation of safety controllers. Although YAP might be more widely used, adaptive and autonomous cyber-physical systems with their highly automated and complex safety mechanisms [2] are in its focus.

Sect. 2 briefly revisits the example discussed in more detail in [6]. Sect. 3 describes preliminaries of YAP. Sect. 4 proposes a workflow to derive a safety controller. Sects. 5.1 to 5.5 detail this workflow in the format of a hands-on guide. Sects. 6 and 7 discuss directions for future work and conclude.

2 Running Example: A Collaborative Manufacturing Robot

We illustrate the proposed workflow by example of a human-robot collaboration (HRC) in a manufacturing work cell with a collaborative robot arm [6]. This work cell has a safeguarded workbench, which is manually supplied with work pieces to be processed by a robot arm and a welder within a safeguarded

[*]This research was funded by the Lloyd's Register Foundation under the AAIP grant CSI:Cobot.

area next to the workbench. The robot moves to the workbench, grabs the work piece, and moves to the welder. The robot and the welder together perform a specific welding task on the work piece. After finishing this task, the robot arm returns the work piece to the workbench where the operator picks it up and supplies the robot with another work piece to repeat this cycle. The work cell is equipped with several safety modes (e.g. safety-rated monitored stop) and safety functions (e.g. a warning display) operated by a safety controller on occurrence of a CE (e.g. operator close to weld spot while welder and robot are working). This way, the safety controller works on top of this cyclic manufacturing process.

3 Overview of Yap: Modelling Concepts and Tool Features

Yap is a research tool for risk modelling, analysis, and design of safety controllers.[1] Yap's input language is a domain-specific language (DSL) providing a corresponding set of modelling primitives.

Activities provide a finite abstraction of the physical process of interest and can be useful for modelling the task structure of an application as well as for structuring a risk analysis accordingly. For example, the task of the robot arm exchanging a work piece can be separated from a welding task performed by the welder and the robot arm.

Risk factors (factors for short) describe the *life-cycle* and constituents of CEs potentially being observed when performing the activities, for example, the robot arm and the operator being on the workbench simultaneously. The hazard list can be modelled as a list of factors. *Factor dependencies* specify temporal or causal relations between risk factors (e.g. `requires`, `prevents`). For example, the fact that the robot arm touches the operator `requires` the operator to be in one of the safeguarded areas.

A factor is modelled by four life-cycle *phases* (i.e., inactive 0^f, active f, mitigated $\overline{f}$, and mishap $\underline{f}$) and five *events* (i.e., endangerment, mitigation, resumption, mishap, alleviation). Four of these events can be refined into *modes* and attributed with (quantitative) *parameters*. For example, the *endangerment* from an operator entering the workbench while the robot is handling a work piece there is *detected* by a light barrier and the robot position. This event can be *mitigated* by a safety-rated monitored stop (`srmst`) and signalling the operator to leave the workbench. After the operator has followed this advice, the robot can *resume* its work piece handling. In case of a detected mishap, a potential consequence could be *alleviated* by a complete shutdown of the work cell and an emergency call. Modes can be embodied by physical and logical *items*, particularly, *actors* (synonymously, agents), constituting the *application*. For example, the logic for the safety-rated monitored stop could be embodied by the robot arm.

These modelling primitives will be explained and used in Sect. 5. A more detailed description of Yap's DSL is, however, provided in [5]. Overall, Yap models can inform controller design by injection of a model of a safety controller into a process model of the application. However, apart from this use case, with Yap's DSL one can describe operational risk as an abstract state machine, explore its symbolic state space, that is, the *risk space*, shape its transition relation, and perform a light-weight symbolic simulation of CE occurrence and handling. Furthermore, one can calculate risk spaces and properties thereof (e.g. mitigation orders [7]), and generate minimal cut sequences [6, 5].

4 Overview of the Workflow

Figure 1 indicates the methodological context in which Yap can be used. There, the specification and synthesis of a safety controller consists of several work steps (1 to 12) in five stages.

[1] Yap, its manual, and the running example can be obtained from `http://www.gleirscher.de/yap/` or from the author.

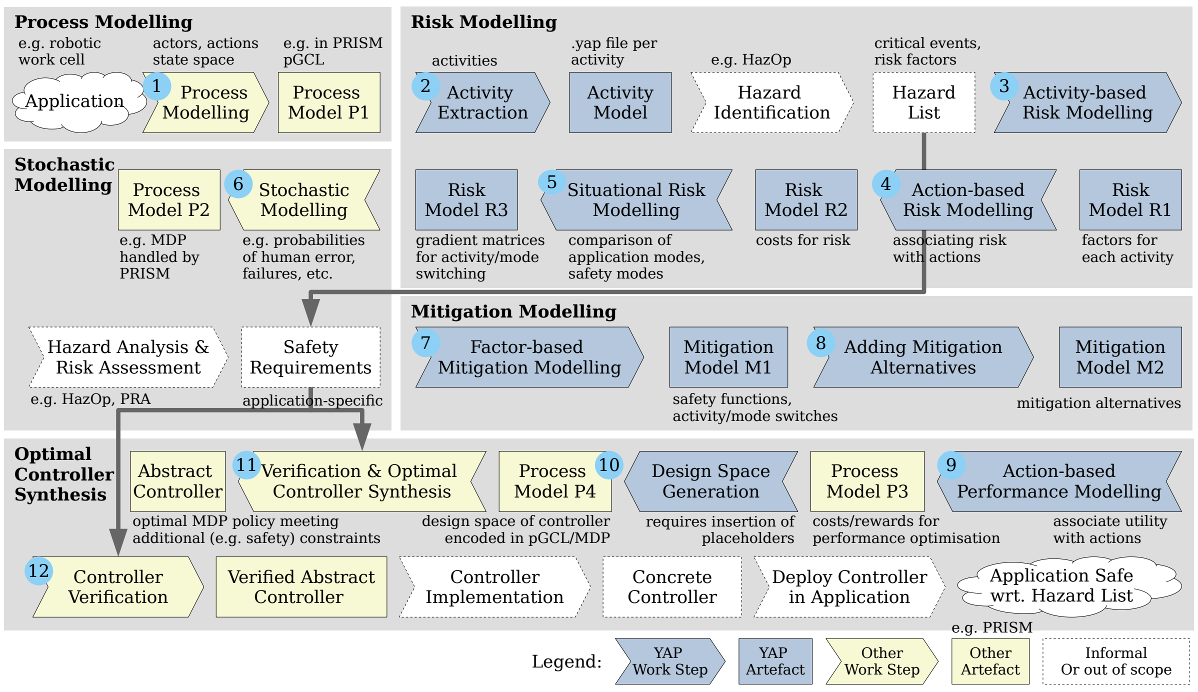

Process Modelling
e.g. robotic work cell
actors, actions state space
e.g. in PRISM pGCL
Application
1 Process Modelling
Process Model P1
Risk Modelling
activities
.yap file per activity
e.g. HazOp
critical events, risk factors
2 Activity Extraction
Activity Model
Hazard Identification
Hazard List
3 Activity-based Risk Modelling
Stochastic Modelling
Process Model P2
6 Stochastic Modelling
e.g. MDP handled by PRISM
e.g. probabilities of human error, failures, etc.
Risk Model R3
5 Situational Risk Modelling
Risk Model R2
4 Action-based Risk Modelling
Risk Model R1
gradient matrices for activity/mode switching
comparison of application modes, safety modes
costs for risk
associating risk with actions
factors for each activity
Mitigation Modelling
Hazard Analysis & Risk Assessment
Safety Requirements
e.g. HazOp, PRA
application-specific
7 Factor-based Mitigation Modelling
Mitigation Model M1
8 Adding Mitigation Alternatives
Mitigation Model M2
safety functions, activity/mode switches
mitigation alternatives
Optimal Controller Synthesis
Abstract Controller
11 Verification & Optimal Controller Synthesis
Process Model P4
10 Design Space Generation
Process Model P3
9 Action-based Performance Modelling
optimal MDP policy meeting additional (e.g. safety) constraints
design space of controller encoded in pGCL/MDP
requires insertion of placeholders
costs/rewards for performance optimisation
associate utility with actions
12 Controller Verification
Verified Abstract Controller
Controller Implementation
Concrete Controller
Deploy Controller in Application
Application Safe wrt. Hazard List
e.g. PRISM
Legend:
YAP Work Step
YAP Artefact
Other Work Step
Other Artefact
Informal Or out of scope

Figure 1: Intended workflow to be used with YAP and the stochastic model checker

Process Modelling. 1 One begins with constructing a behavioural model (P1) of the physical process of interest, focusing on actors, actions, and the state space these actions can modify (Sect. 5.1). For this use case of YAP, we encode this model in the probabilistic guarded command language (pGCL) of the PRISM stochastic model checker [15] for the analysis of Markov decision processes (MDPs). An MDP is a stochastic model, particularly well-suited for reasoning about processes with non-deterministic decisions over actions and probabilistic outcomes of these actions. pGCL offers a concise way of encoding MDPs. PRISM conveniently implements pGCL with parallel composition [8, 21].

Risk Modelling. 2 To structure the risk analysis, we decompose the process into activities (Sect. 5.2). 3 For each activity, we identify risk factors and incidents using an appropriate technique (e.g. the hazard identification stage of hazard operability studies (HazOp) [10]) and capture the results, that is, the hazard list, in a risk model (R1, Sect. 5.2). The practical reasoning and the domain expertise to apply when using a typical system hazard analysis and risk assessment (HARA) technique, are very well-documented in a rich corpus of literature (e.g. [18, 13, 3]). Hence, we touch HARA specifics only selectively. 4 We extend this risk model (R2) by associating to every action a number that encodes the risk of any of the captured incidents from the performance of this particular action (Sect. 5.2). 5 Furthermore, the risk model (R3) is extended by assigning to every transition between two activities or safety modes a number that encodes the increase or decrease in risk of any incident when taking this transition (Sect. 5.2). The result are two *gradient matrices*, one for activity changes and one for safety mode changes.

Stochastic Modelling. 6 We return to the process model and introduce probabilistic phenomena such as human error and sensor failure (Sect. 5.3). As a result, we get a process model (P2) amenable to HARA, for example, further steps of HazOp and, particularly, probabilistic risk assessment (PRA).

Mitigation Modelling. 7 Based on the process and risk models, we design mitigations for each of the identified risk factors (Sect. 5.4). An individual mitigation can perform a safety function, an activity change, and safety mode change, and will after removal of the corresponding risk factor return the process to a state where normal operation can continue. Within YAP, the mitigation model (M1) refines the risk model by an abstract state machine. This state machine is translated into the language used for the process model. 8 YAP allows the definition of alternatives for the mitigation of a single risk factor (Sect. 5.4). The result is a mitigation model (M2) with these alternatives creating a *controller design space*.

Verified Controller Synthesis. 9 Similar to the assignment of risk to process actions, we now associate other costs and rewards (e.g. nuisance of the operator, energy consumption, utility of a performed robot action) with these actions (Sect. 5.5). From this step, we obtain a reward-enhanced MDP model of the application (P3). 10 Next, we use YAP's pGCL generator to translate the risk and mitigation models into a set of pGCL fragments. These fragments fill placeholders easily inserted into the process model beforehand. The resulting process model (P4) is amenable to property verification and acts as a design space for controller synthesis (Sect. 5.5). This design space includes the set of choice resolutions of the MDP as well as further degrees of freedom stemming from other unfixed model parameters (e.g. probabilities 6, rewards 9). We can now verify properties of this design space. 11 Importantly, we use PRISM not only for property verification but also for the selection of a policy (also called adversary or strategy), that is, a particular choice resolution representing the controller, from this design space. Optimal MDP policies are artefacts of quantitative verification and can be represented as discrete-time Markov chains (DTMCs). Depending on the safety requirements, the chosen policy will have to meet certain safety constraints and be Pareto-optimal with respect to the considered performance criteria (Sect. 5.5). 12 We finally verify further safety properties of the policy (Sect. 5.5). Theoretically, 11 and 12 could be collapsed into one verification step carried out on the design space. However, the tooling for this use case requires us to separate Pareto optimisation and constraint verification.

5 Workflow for Controller Design

The following sub-sections provide a hands-on guide detailing the workflow outlined in Figure 1.

5.1 1 Process Modelling: The Physical World

We create a stochastic model (an MDP) of the manufacturing process, as described in Sect. 2, and employ the PRISM model checker for its analysis. We model *actors* (e.g. a robot arm, an operator), *activities* (e.g. exchWrkp for exchanging a work piece, welding a work piece), and (atomic) *actions* (e.g. the robot grabs the work piece, the operator enters the work cell). In PRISM, actors can be implemented as modules and actions as guarded commands of the form `[EVENT] GUARD` $\rightarrow$ `UPDATE`.

Listing 1 exemplifies the actor **robotArm**, participating in the two activities exchWrkp and welding with several actions. For example, the involvement of **robotArm** in exchWrkp is implemented by the three actions r_moveToTable, r_grabLeftWorkpiece, and r_placeWorkpieceRight. The structure of many of the modelled actions follows a specific pattern:

$$\texttt{[ACTOR\_ACTION] !CYCLEEND} \,\&\, \texttt{SM} \,\&\, \texttt{ACTIVITY} \,\&\, \texttt{CUSTOM} \rightarrow \texttt{UPDATE}\,.$$

Action names carry prefixes to indicate the actor(s) performing these actions, that is, **r** for a robot action, **rw** for a compound action of the robot and the welder, **h** for a physical human action, **hi** for an internal

Listing 1: Process model fragment for the robot arm in PRISM

```
module robotArm
reffocc:   bool         init false; // is the grabber occupied?
wpfin:     bool         init false; // is the work piece finished ?
rloc:      [atTable..atWeldSpot]  init inCell; // robot arm location
//<%
...
// exchWrkp: exchange a work piece between workbench and welder
[r_moveToTable] !CYCLEEND & (safmod=normal|safmod=ssmon|safmod=pflim) & ract=exchWrkp & !rloc
    =sharedTbl & ((wps!=right&reffocc)|wps=left&!reffocc) -> (rloc'=sharedTbl);
[r_grabLeftWorkpiece] ...;
[r_placeWorkpieceRight] ...; ...
// welding: carry out welding task together with welder
[r_moveToWelder] !CYCLEEND & (safmod=normal|safmod=ssmon|safmod=pflim) & ract=exchWrkp &
    reffocc & !wpfin -> (ract'=welding)&(rloc'=atWeldSpot);
[rw_weldStep] ...;
[rw_leaveWelder] ...; ...
endmodule
```

human decision, **s** for a synchronous action of the safety controller and other actors, **si** for an independent controller action. `CYCLEEND` is a predicate that, when `true`, terminates model execution. Each action has to be guarded by the activities (`ACTIVITY`) and safety modes (`SM`) it is allowed to be performed in. Action-specific guards (`CUSTOM`) and updates (`UPDATE`) are conjoined and specified. The **robotArm** does not contain stochastic phenomena whereas **humanOp** and other parts of the process model do. The use of probabilistic updates for human errors and other phenomena will be further discussed in Sect. 5.3.

5.2 Risk Modelling with YAP

2 **Activity Modelling.** We create a YAP file for each activity (e.g. generic task or sub-task) of the manufacturing process. For example, for the activity exchWrkp, the listing on the right specifies relationships to other activities. Particularly, exchWrkp inherits (`includes`) attributes (i.e., hazard, activity successors, etc.) from the generic activity moving and can be followed (`successor`) by either of the basic activities off, welding, or idle. With the following command, YAP identifies all activities reachable from the activity off in form of a labelled transition system (LTS). For inspection, this LTS is visualised as a graph in Figure 2.

```
Activity {
  include moving;
  successor welding;
  successor off;
  successor idle;
}
```

```
yapp --global-logging -m off.yap \
     -o output/off-act.dot --showmodel activities .
```

3 **Activity-based Risk Modelling.** We create a hazard list for each activity, for example, by performing a HazOp [10]. Then, we derive factor specifications from these lists. For instance, for the activity welding, we specify five factors, such as the factor "Human arm and Robot on shared Workbench" (HRW, Figure 3). Then, we specify factor dependencies, for example, HRW `requires` the factor "Human arm on Workbench" (HW), `prevents` the activation of factor "Human Close to weldspot" (HC), and HRW's mitigation prevents the mitigation (`mitPreventsMit`) of HC and "Human in Safeguarded area" (HS).

The inclined reader will recognise that HRW is not only a critical event in the process, it also contains the hazard "Robot on shared Workbench". As a whole, HRW can be seen as a latent cause of the incident

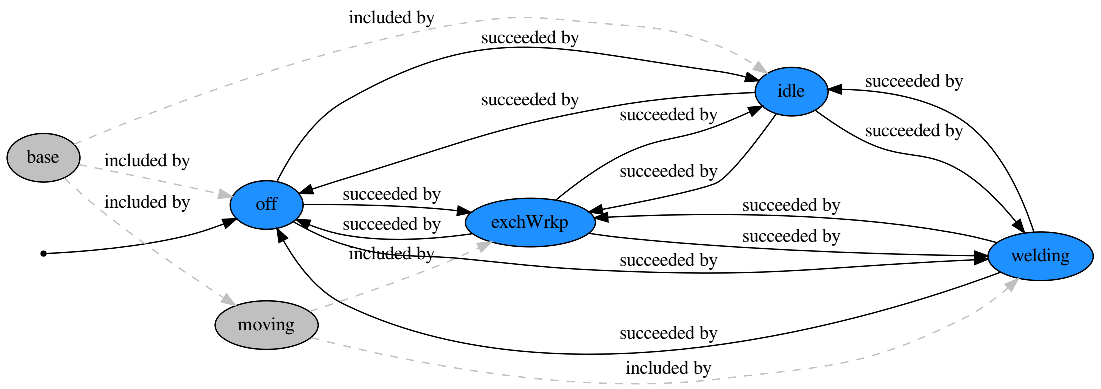


Figure 2: Activity graph showing all activities reachable from the activity off

```
HRW desc "(H)uman arm and (R)obot on
          shared (W)orkbench"
  requires (HW)
  prevents (HC)
  mitPreventsMit (HS,HC) ...
```

```
RT desc "(R)obot arm (T)ouches
         the operator"
  requiresNOf (1|HRW,HS,HC)
  mitPreventsMit (RC)
...
```

Figure 3: Specifications of the two risk factors HRW and RT

"Robot arm Touches the operator" (RT). Accordingly, the specification for RT in Figure 3 uses the constraint `requiresNOf (1|HRW,HS,HC)` to refer to its potential causes: at least one of HRW, HS, and HC.

4 **Action-based Risk Modelling.** In YAP, actions of both the controlled process and the safety controller can be characterised using multiple weights that order the choice alternatives when numerical solvers search the MDP policy space. YAP converts these weights into action reward structures used for quantitative verification in tools such as PRISM.

Like for the actions of the safety controller, one can specify such characteristics for the actions of any actor in the process. This is done in YAP by providing a `weight` structure (Listing 2). For action-based risk modelling, parameters prefixed with `risk_`, say `risk_f` for a factor f, are used to accumulate risk in reward calculations for the underlying MDP. The corresponding action rewards are guarded by f, requiring that f is active. For example, if `weights` provides a `risk_f` entry for the process action `a` then `a` is rewarded only if it is taken in a state where f holds [15].

Connecting the YAP and PRISM Models. Crafting the YAP model and generating a controller model to be integrated into a process model require the engineer to have substantial knowledge of the process model. For example, in order for YAP to inject valid synchronisation commands into existing PRISM

Listing 2: Structure for specifying risk values on a per-action basis

```
Weights rewards {   guard      risk_HC;
   // robotArm
 r_moveToTable:      ""          "5";
 ...
   // welder
 rw_weldStep:        ""          "10";
 rw_leaveWelder:     ""          "5";
   // humanOp
 ...
 h_approachWeldSpot: ""          "7";
 h_exitCell:   "notif=leaveArea" "0"; }
```

Listing 3: Connecting YAP and PRISM models via YAP agents and PRISM modules

```
robotArm type AGENT
  validActs="exchWrkp|welding|off";
welder type AGENT
  validActs="welding|idle|off";
safetyCtr type CONTROLLER
  notif="[ok..resetCtr] init ok"
  notif_leaveWrkb="bool init false";
```

Listing 4: Two risk gradient matrices

```
Distances act {
 off:      0;
 idle:     1 0;
 exchWrkp: 3 2 0;
 welding:  5 4 2 0; }
Distances safmod {
 normal:  0;
 hguid:   -2  0;
 ssmon:   -1  1    0;
 pflim:   -2  0   -1  0;
 srmst:   -3 -1   -2 -1  0;
 stopped: -4 -2  -3 -2 -1 0; }
```

modules, one has to specify the activities the respective actors are involved in.[2] This is done by providing a `validActs` parameter taking values of the form `act1|act2|...|actn`. For example, in Listing 3, the **robotArm** is involved in the three activities exchWrkp, welding, and off, the **welder** in welding, idle, and off. If `validActs` is not provided then YAP assumes that the actor can be involved in any of the specified activities. Furthermore, additional module-specific (i.e., local) variables used by the safety controller can also be declared as part of an item specification.

5 **Situational Risk Modelling.** In YAP, we can not only model risk in terms of factors but also in terms of the behavioural modes the application or controlled process can be in. Examples of such modes are activities and safety modes (Sect. 5.4) as used and standardised in application domains such as HRC [11]. Of course, we allow behavioural modes to change during operation. Moreover, we often do not exactly know about the absolute risk level of a certain mode in a certain situation. Thus, YAP offers the possibility to define *risk gradient matrices* that only capture expected changes in the risk level when changing from one mode into another. We only consider symmetric changes and, thus, use skew-diagonal matrices reduced[3] to their lower left triangles. Examples of these matrices are shown in Listing 4.

To use such matrices for maximisation in YAP's pGCL generator,[4] we associate a positive gradient with an improvement of the risk level and, vice versa, a negative gradient with a worsening of the risk level.[5] For example, the act matrix tells us that a transition from the activity welding to the activity exchWrkp improves the risk level by 2. As a further example, the safmod matrix states that a transition from the safety mode srmst (i.e., safety-rated monitored stop) to the safety mode ssmon (i.e., speed and separation monitoring) worsens the risk level by -2.

YAP uses these matrices to calculate act- and safmod-updates of the set of guarded commands generated for the safety controller from the given factor specifications [6]. In YAP, risk can be evaluated both based on transitions through the risk space (i.e., from one risk state to another) and on a wider situational basis (i.e., based on transitions from one activity or safety mode to another). Note that while in step 4, actions are associated with an *absolute risk* value, in step 5, mode and activity changes are associated with a *risk gradient*, a relative measure stemming from a pair-wise comparison of modes and activities. Whereas for individual actions, the YAP user needs to agree on a single global scale for risk assessment,

[2] That could also be achieved by parsing the process model (here, the PRISM file) but is beyond YAP's current functionality.
[3] For readability, one can provide the symmetric upper right triangle as well. However, YAP internally mirrors the values from the lower left to the upper right to ensure skew-diagonality. [4] A description of YAP's algorithms is out of scope of this tool paper. [5] This choice should not be too counter-intuitive as one can associate negative numbers with something undesirable.

the higher complexity of risk assessment for modes and activities is taken account of by gradients.

5.3 6 Stochastic Modelling

Probabilistic choice in the process model set up in Sect. 5.1 (e.g. an MDP modelled in PRISM's pGCL) can capture a variety of adverse and critical stochastic events, such as human errors, sensor failures, actuator perturbation and failures, and mishaps.

Human Errors. We consider, for example, intrusion of **humanOp**, whether intentional or erroneous, into the work cell when not allowed:

```
[hi_mayEnterCell] !dntFlg_enterCell & !mntDone & !CYCLEEND
    // deontic flag and end-of-cycle check
    & wps!=empty & hact=idle & (hloc!=inCell & hloc!=atWeldSpot) & !wpfin
    // action physically possible / feasible / reasonable
    -> ((prm_enterCell&req_enterCell)?.9:.2):(dntFlg_enterCell'=true)
    // action enabled if (90) / ifnot (10) allowed / required
      +((prm_enterCell&req_enterCell)?.1:.8):true;
    // action notenabled if (10) / ifnot (90) allowed / required
```

This listing shows a two-staged[6] guarded command modelling the operator's internal decision, not their physical action. The flag `dntFlg_enterCell` is *deontic* in the sense that it *enables but not triggers* the action h_enterCell of the operator actually entering the work cell. The activation of this flag is handled by a *conditional probabilistic choice*: the condition `prm_enterCell` distinguishes between probabilistic intrusion in states where the human operator is *permitted* to enter the work cell and probabilistic intrusion where the operator is not allowed to enter. In case of permission, `dntFlg_enterCell` is set to `true` with a 90% chance and, in case of denial, only with a 20% chance. This is a way of saying that the operator commits a human error in 20% of the times *when they should not* enter the work cell. The predicate `req_enterCell` is not used in this particular model. It just indicates another potentially useful state selection mechanism. The condition and the deontic flag can be omitted if the probabilities are universal and we do not need to separate a *physical* action (e.g. movement of the operator) from a *logical* action (e.g. change of the operator's mind). In this model, we treat operating errors and malicious misuse in the same way, however, a distinction can be necessary in other contexts.

Sensor Failures. The separation of physical and logical human actions allows us to synchronise sensor actions with physical actions, for example, to model a range detector with a 5% chance of failure:

```
[h_enterCell] true -> .95:(rngDet'=near)+.05:true;
```

h_enterCell is an event with two *synchronous* actions, the physical action of **humanOp** entering the work cell and the logical action of the range finder in **sensorUnit**. Synchronisation establishes real-time behaviour despite the deontic nature of pGCL. This sensor failure is modelled by the range finder signalling `near` to **safetyCtr** only in 95% of the cases where **humanOp** actually enters the cell.

Actuator Perturbation and Failures. We have not modelled any perturbations in the HRC case study. However, an obvious entry point for such phenomena would be the actions of **robotArm**. For example,

[6] Such commands do not increase the expressiveness of pGCL as they can be expressed by a set of ordinary commands. However, they increase convenience by allowing what can be called conditional probabilistic choice.

the action r_grabLeftWorkpiece could be extended by a probabilistic update modelling the fact that grabbing a work piece fails in a certain fraction of trials with a work piece not in the grabber and/or still in the work piece support, or whatever outcome seems realistic.

Mishaps after Critical Events. Based on conditional probabilistic choice, similar to human error modelling, we use probabilities to capture the fact that from activated critical events and certain actions, a transition into a mishap state is possible. Consider the example:

```
HC desc "(H)uman (C)lose to active welder and robot working"
  ...
  mis="h_exitCell" // mishap possibly initiated by the action h_exitCell
  prob=0.05 // probability of mishap is 5 percent if unrecognised OR active and not mitigated
  sev=5; // severity of the mishap is of class 5
```

The parameter `prop=0.05` defines the probability of a `mishap` from the action h_exitCell in case of an activated HC to be 5%. The parameter `sev=5` associates the impact from such a mishap to the exemplary impact class 5, which can, for example, mean "high". In other words, if **humanOp** wants to perform the action h_exitCell if the factor HC is active (i.e., *HC*) then, with a 5% chance, the outcome will be the mishap $\underline{HC}$, a specimen of the `MISHAP` state. The resulting pGCL fragment generated by YAP:

```
[h_exitCell] true ->
   ((!HCp=mis & (CE_HC | RCE_HC))?0.05:0):(HCp'=mis)
   +((!HCp=mis & (CE_HC | RCE_HC))?0.95:1):true;
[h_placeWorkpieceLeft] true ->
   ((!HRWp=mis & (CE_HRW | RCE_HRW))?0.01:0):(HRWp'=mis)
   +((!HRWp=mis & (CE_HRW | RCE_HRW))?0.99:1):true;
```

$\underline{HC}$ can, for instance, be a welding spark injuring the operator, or the **robotArm** hitting them, both encoded in a corresponding risk/severity reward generated by YAP:

```
[h_exitCell] (!HCp=mis & (CE_HC | RCE_HC)) : 5.0;
```

One downside of this way of reward modelling in PRISM is that the reward is paid independent of the outcome of h_exitCell. A solution not chosen here would be to introduce an intermediate state with the disadvantage of doubling the state space each time such a construction is chosen.

5.4 Mitigation Modelling with YAP

7 **Factor-based Mitigation Modelling.** The basic factor model allows mitigation in one (direct) or two stages (indirect). For controller synthesis, we focus on the two-staged approach, which suggests the specification of modes for detection, mitigation, and resumption.[7] For example, the modes for the factor HRW are referred to from its specification:

```
HRW desc "(H)uman arm and (R)obot on shared (W)orkbench" ...
  guard "hACT_WORKING & rloc=sharedTbl & hloc=sharedTbl"
  detectedBy (SHARE.HRWdet)
  mitigatedBy (PREVENT.HRWmit)
  resumedBy (.HRWres) ...
```

and detailed in the `Application` fragment:

```
mode HRWdet
  guard "hACT_WORKING & rloc=sharedTbl & lgtBar=true"
  embodiedBy cellObSys;
```

[7] We omit alleviation modes for the sake of simplicity of this guide.

```
mode HRWmit desc "safety-rated monitored stop"
  update "(notif_leaveWrkb'=true)" // safety function on
  target (safmod=srmst) // safety mode on
  embodiedBy robotArm
  disruption=5 nuisance=1 effort=0.5;
mode HRWres
  guard "hloc!=sharedTbl" // checking for hazard removal
  update "(notif_leaveWrkb'=false)" // safety function off
  target (safmod=normal); // safety mode off
```

Detection, mitigation, and resumption modes make use of the process model. For example, HRWdet uses the formula `hACT_WORKING` (explained below) and the module variables `rloc` and `hloc` (the locations of the robot arm and the operator) in an embedded `guard` expression. `update` in HRWmit specifies the assignment of `true` to the module variable `notif_leaveWrkb` to notify the operator to leave the workbench. Being a generic form of update, `target` specifies mode switching preferences that, when unsafe, will be overridden by YAP using the gradient matrices (Listing 4).

For example, HRWmit switches to the mode "safety-rated monitored stop" (`smrst`) and HRWres back to the `normal` mode. However, **safetyCtr** would only switch to the `normal` mode if this is acceptable in the risk state to be reached. Hence, each of these modes will be enhanced by information about activities and risk states and translated into one or more guarded commands for being integrated into the process model. Note that there are also *embodiment* references to the items **robotArm** and **cellObSys**, the latter being the overall sensor system of the work cell.

Connecting the YAP and PRISM Models. It is convenient to reuse formulas in several places. For this purpose, YAP allows their definition in the `Application` sections of the YAP model:

```
Application cobot {
  hACT_WORKING = "(ract=exchWrkp | ract=welding | wact=welding) & safmod=normal";
  ...
  hST_HOinSGA = "hloc=inCell | hloc=atWeldSpot";
  hFINAL_CUSTOM = "wpfin & wps=empty & !reffocc & mntDone"; ... }
```

For example, the predicate `hACT_WORKING` includes activities where actors are effectively working. `hST_HOinSGA` is a shortcut specifying states where the operator is in the safeguarded area. `hFINAL_-CUSTOM` refines `CYCLEEND` (Sect. 5.1), the termination of a process cycle, in our example, the end of a manufacturing cycle in the work cell. This predicate is used in the reduction of cyclic end components in order for optimal MDP policy search algorithms to work correctly [15].

8 Adding Mitigation Alternatives. Factor specifications allow one to provide several mitigation and resumption options and characterise their properties using *risk and performance* parameters. For example, the factor HS (Human in Safeguarded area while robot working or welding) refers to three mitigation options in its `mitigatedBy` directive:

```
HS desc "(H)uman in (S)afeguarded area while robot working or welding"
  guard "hACT_WORKING & (hloc=inCell | hloc=atWeldSpot)"
  detectedBy (SHARE.HSdet)
  mitigatedBy (.ssmon,.srmst,.stopped)
  resumedBy (.HSres)
```

We model the options ssmon, srmst, and stopped in more detail in Listing 5.

Listing 5: All modes including for the factor HS three mitigation options

```
mode HSdet desc "range detector"
  guard "hACT_WORKING & (rngDet=near |
      rngDet=close)";
mode ssmon
  desc "speed/separation monitoring"
  target (safmod=ssmon)
  disruption=9 nuisance=9 effort=8;
mode srmst
  desc "safety/rated monitored stop"
  target (safmod=srmst)
  disruption=5 nuisance=9 effort=5;
mode stopped
  desc "protective emergency stop"
  target (safmod=stopped)
  disruption=2 nuisance=6 effort=3;
mode HSres
  guard "!hST_HOinSGA" // check hazard removal
  target (safmod=normal);
```

Table 1: Placeholders recognised by YAP and to be inserted into the process model for substitution

Placeholder	Description
`<%YAP#TYPES>`	Inject global type declarations
`<%YAP#PREDICATES>`	Inject global definitions
`<%YAP#CONTROLLER>`	Inject controller module
`<%YAP#REWARDS>`	Inject reward structures
`<%YAP#MODULEHOOK(m)>`	Add data and command definitions to application module `m`

5.5 Verified Controller Synthesis

9 **Action-based Performance Modelling.** Along with the three mitigation options specified in Listing 5, we provide estimates for their *disruption* of the manufacturing process, for their *nuisance* of operators, and for the *effort* required for their execution. In addition, for each action in the process model (Sect. 5.1), one can provide a *guard* and several columns of optimisation parameters (e.g. `prod`, `eff_process_time`, `risk_HC`). Each such column is converted into an action reward structure.

```
Weights rewards {
           guard   prod  eff_process_time;
  // robotArm
 r_moveToWelder: ""        "h"    "2*macro";
 ...
  // welder
 rw_weldStep:   ""         "h"    "3*macro";
 rw_leaveWelder: ""        "h"    "macro";
  // humanOp
 h_start:    ""          "1"    "macro";
 ...
 h_enterCell:    ""         "none" "none";
 h_exitCell: "notif=leaveArea""none""none";
}
```

Moreover, as shown in these two examples, one can also use parameters defined elsewhere (e.g. `macro, h, none`) instead of using literal numbers in place. One may provide several `weights` structures across the activity model. However, they will all be merged into one central "database" containing all columns found in the given structures.

10 **Design Space Generation.** The process model has to be instrumented with *placeholders* to be substituted with model fragments generated by YAP. Such placeholders take the form `<%YAP#`*X*`>` where *X* is the placeholder name.[8] Table 1 lists placeholders currently supported by YAP.

The output of YAP, for this use case, is an MDP, with its non-deterministic choice representing the decision space of all actors in the work cell. We call the decision space of the safety controller—as one

[8] The placeholders will need to be commented in order to not interfere with the semantics of the process modelling language (cf. Javadoc in Java). In PRISM, we therefore use `//<%YAP#`*X*`>`.

of these actors—the *design space*. This design space and the decision space of the other actors are used for optimal controller synthesis.

With the model constructed according to the steps 1 to 9 in Figure 1, we use YAP to generate and inject the controller into the *process model* (Sect. 5.1):

```
yapp -m model.yap -t target-template.xyz -o output/model.xyz \
     -f prism -d multi-event-concurrent --synthesise controller
```

With the output format switch `-f prism`, YAP creates three artefacts in this step:

1. An MDP in PRISM's pGCL used as the *design space* for PRISM's search for an *optimal policy*—a DTMC—representing the abstract controller (file `model.prism`).
2. A list of probabilistic computation tree logic (PCTL) properties to be verified of the design space by PRISM in step 11 (Sect. 5.5, file `model.props`).
3. A list of PCTL properties to be verified in step 12 (Sect. 5.5) of any policy found by PRISM (file `model_pol.props`).

The following pGCL fragment, generated by YAP, shows the design space for *switching into a safety mode* triggered if a particular hazard (e.g. HC, HRW, HS) has been detected:

```
[si_HCSrmstIdleVissafmod] !CYCLEEND & safmod=normal & HCp=act -> (safmod'=srmst);
[si_HCStOffAudsafmod] !CYCLEEND & safmod=normal & HCp=act -> (safmod'=stopped);
[si_HCStOffVissafmod] !CYCLEEND & safmod=normal & HCp=act -> (safmod'=stopped);
[si_HRWmitsafmod] !CYCLEEND & safmod=normal & HRWp=act -> (safmod'=srmst);
[si_srmstsafmod] !CYCLEEND & safmod=normal & HSp=act -> (safmod'=srmst);
```

The following pGCL fragment, generated by YAP, highlights the part of the controller design space allowing the controller to *switch off mode-specific safety functions* (lines 1-2) and *resume to a less restrictive safety mode* (lines 3-4):

```
1 [si_HCres2fun] !CYCLEEND & HCp=mit & !CE_HC & notif=leaveArea & !hST_HOinSGA -> (notif'=ok);
2 [si_HCresfun] !CYCLEEND & HCp=mit & !CE_HC & notif=leaveArea & !hST_HOinSGA -> (notif'=ok);
3 [si_HRWressafmod] !CYCLEEND & safmod=normal & HCp=inact & HSp=act & WSp=inact & HWp=inact &
      RCp=inact & (RTp=mit | RTp=sfd) & HRWp=mit & hloc!=sharedTbl & (notif_leaveWrkb=false)
4     -> (safmod'=ssmon)&(HRWp'=sfd);
```

11 Design Space Verification and Optimal Controller Synthesis. In this example, we use PRISM for the verification of the MDP and the synthesis of an MDP policy. The use of PRISM and the processing of its output is out of scope of this paper and, therefore, not explained here. Particularly, the formulas presented below contain PCTL operators [14] and other PRISM query language [21] primitives that are assumed to be familiar to the YAP user interested in PRISM-based controller synthesis with YAP.

Synthesising Optimal Policies from an MDP. For optimal policy synthesis (here, the synthesis of DTMCs), we use the command

```
prism output/model.prism -pctl '<query>' -s \
     -exportadvmdp poloutdir/model-adv.tra \
     -exportstates poloutdir/model-adv.sta \
     -exportprodstates poloutdir/model-adv.pst \
     -exportlabels poloutdir/model-adv.lab .
```

Table 2: Formulas generated by YAP for custom property specification

Formula	Description
E	"detector" predicate for the critical event E
`RCE_`E	"ground truth or reality" predicate for the critical event E
`ANYOCC (OCE)`	*universal* detector predicate, true if *any* critical event is true
`ANYREC (RCE)`	universal ground truth counterpart of `ANYOCC`
`ANY (CE)`	true if any CE has occurred whether or not detected
`ACCIDENT`	true if any factor f is in its mishap phase $\underline{f}$
`MISHAP`	true if `ACCIDENT` or if any `final` factor (e.g. an incident) is activated (i.e., in phase f)
`SAFE`	true of any state that is neither a CE nor a mishap
`FINAL (CYCLEEND)`	true if `hFINAL_CUSTOM` (Sect. 5.4) is reached

PRISM will generate one or more policy files (with names and suffixes according to `model-adv[1-n].pst,sta,tra,lab`) from the file `output/model.prism` and compliant with the optimisation query `<query>`, for example,

```
multi(R{"effort"}max=? [ C ], R{"nuisance"}max=? [ C ]) ,
```

and places these files in `poloutdir/`. This query searches for all policies that Pareto-maximise effort and nuisance (`Rmax[C]`) as explained in Sect. 5.5. PRISM enumerates such policies as a list of value pairs holding the results of the cost function defined by this query. Within PRISM's GUI, one can visualise these pairs as a Pareto front. We calculated a Pareto front for the present example in [6].

To include the verification of safety properties at this stage in the procedure, we can use a combination of a single optimisation query and several constraints, for example,

```
multi(R{"prod"}max=? [ C ], R{"risk_sev"}<=s [ C ]) and
multi(R{"risk_sev"}<=s [ C<=t ], P<=p [ F "ANY" ]) .
```

The first property maximises the `productivity` of the work cell (`Rmax[C]`) as long as the accumulative bound on action rewards (`R<=s[C]`) for `risk_sev` stays below a user-defined level s. The second property combines a time-bounded version (`C<=t`) of the latter constraint with the probability-bounded reachability (`P<=p[F]`) of `ANY` of the modelled factors, for user-defined bounds t and p. Beyond `ANY`, YAP generates further shortcut formulas (into the file `model.prism`) that can be used in PCTL properties as shown above. These state formulas are listed in Table 2.

12 Controller Verification: Checking the Generated Policies (DTMCs). As already mentioned in Sect. 4, the separation into the two verification steps 11 and 12 and the corresponding property lists is due to restrictions in the combinations of properties that can be checked by PRISM in one go. The policies are given in the form of DTMCs and can, thus, be further checked with the command

```
prism -importstates poloutdir/model-adv.sta \
      -importlabels poloutdir/model-adv.lab \
      -importtrans poloutdir/model-adv1.tra -dtmc \
      -pctl "<prop>" -gs >poloutdir/model-adv-checks.txt .
```

For example, for `<prop>` we applied

```
filter(avg, P=? [ !"ACCIDENT" W "SAFE" ], "ANYREC" & !"MISHAP")
```

to determine the average probability (`filter(avg, P=?  [...], ...)`) of accident freedom until reaching a safe state (`!"ACCIDENT" W "SAFE"`) when starting from any reachable hazardous state (`"ANYREC" \& !"MISHAP"`). As already mentioned in Sect. 5.5, with the file `model.props`, YAP suggests a range of properties to be checked of a policy. See [6, Tab. III] for a selection of properties.

Further Processing of the MDP and DTMC. The abstract controller consists of the list of states of the MDP respectively the DTMC, for example,

```
State:(...  HRWp, notif_leaveWrkb, ...)
510:  (...  1, false, ...)
511:  (...  1, true, ...)
```

and all transitions describing the decisions (accordingly, with probabilistic outcomes), for example,

```
510 511 1 si_HRWmitfun
```

This transition, going from state 510 to 511 and labelled with the action `si_HRWmitfun`, notifies the operator, with probability 1,[9] to leave the workbench in case of an activated factor HRW.

For visualisation and model debugging, the MDP can be converted into a `dot` file with

```
  prism output/model.prism -exporttransdotstates rel.dot .
```

This file can be used with GraphViz tools such as `dot`.[10] Based on GraphViz, YAP provides rudimentary facilities for the visualisation of the generated policy. Such a visualisation can be useful in the direct debugging of DTMCs with a state space of up to a size of around 1000 states.

6 Discussion and Outlook

From Abstract to Concrete Policies. The safety controller in its abstract form is represented by the calculated policy, a DTMC with state space Σ and action set A. Σ and A are results of combining the risk state space, generated by YAP from the factor set F, and the mitigation actions, filed in the YAP model, with the process model. Each state-based memory-less deterministic policy π can then be represented by a map $\pi\colon \Sigma \to A$. As shown before, the transition relation of the policy is provided by PRISM as a list of $(state, action, probability, state)$-tuples (cf. Sect. 5.5). The safety controller, part of such a policy, is a list of transitions that, at the concrete level, would again be guarded commands of the form:

$$\underbrace{[\text{controller action}]}_{\text{event}}\ \underbrace{\text{process \& risk state}}_{\text{guard}} \to \underbrace{\text{mode \& activity switch, safety function}}_{\text{update}}$$

PRISM's output can be used to translate this abstract policy representing the discrete-event controller into a concrete policy. This translation involves two essential steps:

- The translation of the abstract states into concrete guard conditions, and
- the translation of the updates into low-level procedures generating control inputs to the process.

Part of ongoing research is the corresponding refinement of this transition relation into an automaton that can run on, for example, an autonomous machine platform or the robot operating system[11] (ROS). We will investigate how environments such as Isabelle/UTP [4] and ROBOTOOL [20] can be used to verify and deploy safety controllers derived with the help of YAP. Isabelle/UTP provides a generic framework for model verification and ROBOTOOL an environment for rigorous robotic software development.

[9] The controller in this example is fully deterministic but, conceptually, we can also design randomised controllers with our approach. [10] See `https://graphviz.org`. [11] See `https://www.ros.org`.

Safety Properties and Safety Controller. A safety property states that "something will *not* happen" and, thus, is a property whose violation can be observed in finite time [16]. In many applications, "something" refers to a CE that cannot be avoided by a careful redesign of the process and, therefore, violations have to be accepted to a certain extent. A safety controller typically includes a safety monitor responsible for detecting such violations at run-time [17] and an active component influencing the monitored process in a way that the safety property is established again. In other words, the "violation counter" is restored. In such applications, we therefore substitute the verification of the original safety property by

- a response property [19] (formalising successful mitigation and resumption as a finite response to the detection of an endangerment) to be verified of the process integrated with the controller design space (cf. 11), and
- another safety property (cf. 12, formalising the absence of undesired consequences of the aforementioned violations) whose probability of being violated must not exceed a certain bound, by virtue of the safety controller when working correctly.

Re-Interpretation of Activity Graphs for Synthesis. We may want to allow several actors to concurrently carry out actions in any of the activities of the process. Therefore, it seems useful to associate a coloured Petri net (CPN) [12] semantics to activity graphs (e.g. Figure 2). CPNs offer a more flexible way of modelling concurrency compared to the parallel composition [8] used in PRISM's pGCL. Specifically, in a CPN, the places could represent activities and the movable labels the actors. Then, a placement of these labels, that is, a marking, indicates the activities actors are performing at a point in time. A transition in a CPN can move any number of labels between the activities, meaning the actors involved in that transition concurrently finish their current activities and start new activities. However, a pGCL guarded command in PRISM can either move one label or as many labels as there are actors participating in a synchronous event. As a part of our future work, we will investigate how the explicit approach to concurrency in CPNs improves the usefulness and flexibility of the activity model.

7 Conclusions

This paper provides a hands-on and tool-focused guide to a novel approach to the design, verification, and synthesis of safety controllers from hazard analysis and risk assessment as previously published in [6]. We also discuss a range of modelling decisions (e.g. identifying parameters, decomposing behaviour, integrating probabilistic choice) to be made when devising such controllers. The proposed step-wise and tool-supported workflow aims at supporting verification engineers in transforming data from hazard analysis and risk assessment into a verifiable controller model and, thus, contributes to recent and practically relevant challenges (e.g. [1, Challenges OC1, OC2, and OC4]).

Among the next steps of technical research are the improvement of the synthesis facilities, the evaluation of alternatives to PRISM, and the development of an integration with robotic platforms (e.g. ROS, digital twin environments[12]) and tools (e.g. ROBOCHART[20]) to automate controller deployment.

Acknowledgements. This research was funded by the Lloyd's Register Foundation under the Assuring Autonomy International Programme grant CSI:Cobot. I am greatly indebted to Radu Calinescu for many inspiring discussions and for encouraging me to implement some of the described YAP enhancements.

[12] See, e.g. `https://github.com/douthwja01/CSI-cobotics`.

References

[1] Radu Calinescu, Javier Camara & Colin Paterson (2019): *Socio-Cyber-Physical Systems: Models, Opportunities, Open Challenges*. In: *5th ICSE Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, IEEE/ACM, pp. 1–6, doi:10.1109/sescps.2019.00008.

[2] Radu Calinescu, Danny Weyns, Simos Gerasimou, Muhammad Usman Iftikhar, Ibrahim Habli & Tim Kelly (2018): *Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases*. *IEEE Transactions on Software Engineering* 44(11), pp. 1039–1069, doi:10.1109/tse.2017.2738640.

[3] Clifton A. Ericson (2015): *Hazard Analysis Techniques for System Safety*, 2 edition. Wiley.

[4] Simon Foster, Frank Zeyda & Jim Woodcock (2015): *Isabelle/UTP: A Mechanised Theory Engineering Framework*. In: *UTP*, Springer, pp. 21–41, doi:10.1007/978-3-319-14806-9_2.

[5] Mario Gleirscher (2020): YAP *Against Perils: Application Guide and User's Manual*. University of York and Technical University of Munich. Available at `http://gleirscher.de/yap/`.

[6] Mario Gleirscher & Radu Calinescu (2020): *Safety Controller Synthesis for Collaborative Robots*. In: *Engineering of Complex Computer Systems, 25th International Conference (ICECCS), 28 - 31 October 2020, Singapore*, pp. 1–12. Available at `https://arxiv.org/abs/2007.03340`. In press.

[7] Mario Gleirscher, Radu Calinescu & Jim Woodcock (2020): *Risk Structures: A Design Algebra for Risk-Aware Machines*. Working paper, Department of Computer Science, University of York, York, UK. Available at `https://arxiv.org/abs/1904.10386`.

[8] Charles A. R. Hoare (1985): *Communicating Sequential Processes*. Int. Series in Comp. Sci., Prentice-Hall. Available at `http://www.usingcsp.com`.

[9] IEC 61508 (2011): *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. Standard, The 61508 Association. Available at `http://www.61508.org/`.

[10] IEC 61882 (2016): *Hazard and operability studies – Application guide*. Standard 61882, IEC. Available at `https://webstore.iec.ch/publication/24321`.

[11] ISO/TS 15066 (2016): *Robots and robotic devices – Collaborative robots*. Standard, Robotic Industries Association (RIA). Available at `https://www.iso.org/standard/62996.html`.

[12] Kurt Jensen & Lars M. Kristensen (2009): *Coloured Petri Nets*. Springer, Berlin Heidelberg, doi:10.1007/b95112.

[13] John Knight (2012): *Fundamentals of Dependable Computing for Software Engineers*. Chapman and Hall/CRC, doi:10.1201/b11667.

[14] Marta Kwiatkowska, Gethin Norman & David Parker (2007): *Stochastic Model Checking*. In M. Bernardo & J. Hillston, editors: *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM)*, *LNCS* 4486, Springer, pp. 220–70, doi:10.1007/978-3-540-72522-0_6.

[15] Marta Kwiatkowska, Gethin Norman & David Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: *23rd International Conference on Computer Aided Verification (CAV)*, *LNCS* 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[16] Leslie Lamport (1977): *Proving the Correctness of Multiprocess Programs*. *IEEE Trans. Software Eng.* 3(2), pp. 125–43, doi:10.1109/TSE.1977.229904.

[17] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *Journal of Logic and Algebraic Programming* 78(5), pp. 293–303, doi:10.1016/j.jlap.2008.08.004.

[18] Nancy G. Leveson (2012): *Engineering a Safer World: Systems Thinking Applied to Safety*. Engineering Systems, MIT Press, Cambridge, Mass., doi:10.7551/mitpress/8179.001.0001.

[19] Zohar Manna & Amir Pnueli (1995): *Temporal Verification of Reactive Systems: Safety*. Springer, doi:10.1007/978-1-4612-4222-2.

[20] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis & Jim Woodcock (2019): *RoboChart: modelling and verification of the functional behaviour of robotic applications*. Software & Systems Modeling, doi:10.1007/s10270-018-00710-z.

[21] Dave Parker, Gethin Norman & Marta Kwiatkowska (2019): *PRISM Model Checker*. Available at `http://www.prismmodelchecker.org/manual/`.

A Formal Model for Quality-Driven Decision Making in Self-Adaptive Systems

Fatma Kachi   Chafia Bouanaka   Souheir Merkouche
LIRE Laboratory
University of Constantine2-Abdelhamid Mehri
Constantine, Algeria
(fatma.kachi, chafia.bouanaka, souheir.merkouche)@univ-constantine2.dz

Maintaining an acceptable level of quality of service in modern complex systems is challenging, particularly in the presence of various forms of uncertainty caused by changing execution context, unpredicted events, etc. Although self-adaptability is a well-established approach for modelling such systems, and thus enabling them to achieve functional and/or quality of service objectives by autonomously modifying their behavior at runtime, guaranteeing a continuous satisfaction of quality objectives is still challenging and needs a rigorous definition and analysis of system behavioral properties. Formal methods constitute a promising and effective solution in this direction in order to rigorously specify mathematical models of a software system and to analyze its behavior. They are also largely adopted to analyze and provide guarantees on the required functional/non-functional properties of self-adaptive systems. Therefore, we introduce a formal model for quality-driven self-adaptive systems under uncertainty. We combine high-level Petri nets and plausible Petri nets in order to model complex data structures enabling system quality attributes quantification and to improve the decision-making process through selecting the most plausible plans with regard to the system's actual context.
Keywords: Formal methods, Petri nets, Self-adaptive systems, Quality-driven systems, uncertainty models.

1 Introduction

Modern advanced software systems are required to perceive important structural and dynamic changes to their operational environment as well as to their internal status, and to adapt to these continuous changes autonomously [18]. They aim to achieve better quality of service and ensure the required functionality. However, such systems are expected to deal seamlessly with different types of uncertainty during operation. These uncertainties are often difficult to predict at design time, requiring software to be deployed with incomplete knowledge and handle changing conditions during operation [28]. Consequently, software engineers are investigating new techniques to handle uncertainty at runtime without incurring penalties, which is commonly referred to as self-adaptation [18, 12]. Many software systems actually need to comply with strict requirements, providing guarantees for system properties such as ensuring a certain level of performance and reliability.

Self-adaptability [16] is a well-established approach for modelling such systems, and allowing them to be able to achieve functional and/or quality of service objectives by autonomously modifying their behavior at runtime. The MAPE-K [10, 1] (Monitor-Analyze-Plan-Execute over a shared Knowledge) loop is one popular approach that has proven its efficiency in modelling self-adaptability since it covers the necessary activities to be performed in a control loop. However, uncertainty is a fundamental challenge of SASs (Self-Adaptive Systems). It involves not only system requirements but also its execution context and affects the system quality of service. Therefore, self-adaptation mechanisms driven by quality modify system behavior dynamically. Albeit there has been extensive research to address uncertainty in SASs [24, 19, 21], there is no focus on proposing solutions to identify uncertainty at different levels of the decision-making process and considering it when modelling the SASs. Besides, the main focus has been on achieving adaptations without determining their side effects on the overall system qualities. Moreover, existing approaches do not allow choosing the most appropriate adaptation plan in terms of the best side effects.

The engineering of quality-driven self-adaptive systems, evolving under uncertainty, needs to consider the above issues. Besides, designing this type of systems requires a verification and validation phase using formal methods and tools so that the model can be analyzed with regard to quality. Formal methods constitute a promising and effective solution to rigorously specify mathematical models of a software system and analyze its behavior. They are also largely adopted to analyze and provide guarantees on the required properties of self-adaptive systems. In this field, Petri nets [27] have shown their ability as a powerful tool for modelling and verifying complex systems, a number of variants have also been introduced to help modelling uncertainties, such as fuzzy Petri nets [20, 29], possibilistic Petri nets [17], etc.

In this paper, we introduce a formal model for self-adaptive systems evolving in dynamic environments and execution contexts. We mainly leverage a MAPE-K loop-based model, combining high-level Petri nets (HLPNs) and plausible Petri nets (PPNs), for the design of quality driven systems under uncertainty. Although PPNs allow managing uncertainty and guiding the decision-making process, they do not support the complex data structures that are necessary to enable the quantification of system quality properties. To this end, we exploit the expressive power of HLPNs. We mainly extend the model proposed in [5] to capture uncertainty by combining PPNs with the already used HLPNs. The basic approach does not consider the uncertainty concerns in the decision-making process. It does not also allow representing complex information and characteristics of a dynamic system owing to the use of ordinary Petri nets. In addition, the separation of concerns between system qualities may lead to negative side effects. Therefore, we use HLPNs to represent complex systems and data structures; and PPNs to assist the decision-making phase in selecting the best plans in the presence of uncertainty.

The remainder of this paper is structured as follows. Section 2 recalls basic concepts on Petri net types involved in our model. Section 3 discusses related work on modelling SASs and tackling the problem of uncertainty. Section 4 formally describes the proposed model and its components, the latter is illustrated and validated through the problem of the aircraft planning in Section 5. Finally, Section 6 concludes the paper.

2 Background

Petri nets have been initially proposed to model the behavior of a dynamic system with discrete events [6]; they have then undergone several evolutions and variants [15, 9, 2, 26] to cover more concerns in system modelling and analysis. In what follows, we present two types of Petri nets to be used in this work, high-level Petri (HLPNs) nets and plausible Petri nets (PPNs). To facilitate the understanding of our approach and the usefulness of Petri nets types to be adopted, we give only an informal description of what they are and their purpose; we refer the readers to [15, 8] for formal definitions and more details.

2.1 High-Level Petri Nets (HLPNs)

HLPNs are a well-defined semi-graphical technique for the specification, design and analysis of systems. HLPNs are applicable to a wide variety of concurrent discrete event systems and in particular distributed ones. Generic fields of HLPN application domains include: requirements analysis; development of specifications; modelling business and software processes; simulation of systems to increase confidence; formal analysis of the behavior of critical systems; development of Petri net support tools, etc. [15].

An HLPN is made up on a set of nodes (i.e. places and transitions) and a set of arcs connecting places to transitions and vice versa as in an ordinary Petri net but is extended in the following way: each place is associated with a place type and can contain a collection of tokens corresponding to that place type. Transitions are dotted with boolean expressions (for example, $x < y$) called guards. Additionally, arcs are inscribed with expressions called arc annotations; expressions may contain constants, variables, and function images. An expression is evaluated by assigning values to each of its variables. Whenever an expression evaluates to true, a multi-set of tokens is produced in the output places of the corresponding transition according to arcs weights and types.

2.2 Plausible Petri Nets (PPNs)

The combination of the principles of Petri nets with the foundations of information theory resulted in a new model for Petri nets, called plausible Petri nets (PPNs) [7, 8]; which are a hybrid variant of Petri nets composed of two types of places and transitions, namely, symbolic and numerical, in order to describe both discrete and continuous behaviors of a system. In the symbolic subnet, the discrete behavior is described using regular tokens, while in the numerical subnet, continuous or numerical behavior is described with tokens that carry information about the states of variables. A state of information about a given variable is the probability density function (PDF) of x over χ [22], where χ is the state space of a stochastic variable x. For a numerical transition, it can fire when the conjunction between all its input places states of information and the transitions states of information is possible. For a mixed transition, both conditions of symbolic and numerical transitions must be satisfied. Firing transitions effect for the numerical places is a state of information consisting of a disjunction of the previous state of information, and the information produced after firing the transition (conjunction of state of information within the transition and its input places). For more details on PPNs, the reader is referred to [9].

The main feature of PPNs resides in their efficiency to jointly consider the evolution of a discrete event system together with uncertain information about the system state using states of information [8]. They provide a mapping between the possible numerical values of a state variable and their relative plausibility, hence giving greater versatility for representing uncertain knowledge using a more principled approach [9].

3 Related work

Over the past years, researchers have developed a large body of work to formally model self-adaptive software systems and many approaches have shown remarkable progress in providing solutions to mitigating uncertainties in these systems using various formal approaches.

In [5], the authors proposed a formal framework, based on HLPNs, to model a distributed self-adaptive system. In particular, the framework shows how the most significant concepts related to self-adaptation can be formally specified in terms of HLPN, and how structural changes implemented by multiple control loops can be described in a natural way. A two-layered architecture is adopted; it ensures a clear separation between the managed and the managing systems. The HLPN emulator is capable of representing the system dynamics; for this purpose, the basic Petri net is encoded in the emulator marking. The managing system is defined by a collection of MAPE-K control loops, specified using HLPNs. To implement sensors and actuators, the read and write API primitives are used. The approach is consistent and based on well-established formal methods. As such, it takes advantage of consolidated analytical techniques. However, a major drawback of this approach is that it does not consider uncertainties to be faced in the decision-making phase. This leads to poor system performance. Hence when choosing an adaptation plan, the system lacks global knowledge changes to be performed and therefore is unable to determine the negative effect of each plan in order to choose the best one. Besides, it does not allow modelling complex systems due to the use of ordinary Petri nets for modelling the managed systems.

In [24], an approach called Simplex Control Adaptation (SimCA*) is presented, it allows building self-adaptive software systems that satisfy multiple STO-reqs a combination of S-reqs (stakeholder requirements), T-reqs (threshold requirements), and O-reqs (optimization requirements) in the presence of different types of uncertainty. SimCA* has dedicated specific components to monitor changes in the underlying system or its environment and adjust the adaptation logic accordingly to deal with different types of uncertainty. The main contribution of SimCA* is in applying formal techniques to adapt the behavior of software systems, which is one key approach for providing correctness guarantees. Formal analysis in SimCA* is based on an equation-based model of the software system and leverages on guarantees provided by basic SimCA [23]. This analysis is complemented by an empirical evaluation that demonstrates that SimCA* achieves the required quality goals. Although this approach deals with uncertainty and offers correctness guarantees, it does not support systems and uncertainties modelling.

In [9], the authors provide a framework for modelling self-adaptive expert systems (SAeSs) using Petri nets. The Petri nets used here are called plausible Petri nets combined with Bayesian learning principles. PPNs model uncertainty through information states, which provide a map between the possible numerical values of a state variable and their relative plausibility. This methodology is primarily used in SASs that deal with uncertainty, for monitoring system infrastructure assets. This model allows systems to handle uncertainties and to adapt to their occurrence; applying Bayesian learning particularly allows generating new adaptation decisions, which is a suitable solution for the adaptation problem. However, since it uses symbolic subnet which accounts for the discrete behavior of the system using regular tokens, as for ordinary Petri nets, it does not allow representing complex structured data and thus complex systems.

In [14], a new type of Petri net based on neural networks to model adaptive software systems was presented. It is an extension of hybrid Petri nets by embedding a neural network algorithm into them on particular transitions; system adaptation is realized through the learning ability of neural networks. The proposed model considers the runtime environments and ensures that components collaborate to make the suitable adaption decisions while the computing happens locally. We highlight that the model is more efficient than traditional optimization solutions since it is able to not only process runtime data and make decisions, but also model the behavior of software systems. A major drawback of the proposed model is that it does not allow new decisions to be made, i.e. adaptation actions have to be defined statically, in addition to the conditions for their selection, which is a difficult and costly task in the case of complex systems.

Although some of the models presented in this section inspired us to define our model, none of them allow both modelling self-adaptation and managing uncertainty at the same time. So, we can conclude that:

- Models dedicated to self-adaptive systems modelling do not generally allow the generation of new decisions, but rather the decisions are encoded beforehand and the system deduces which one to apply according to the actual context changes. Thus, when new events occur, the system does not know what to do. Moreover, the system's external environment is generally discarded, since in most cases only system resources are considered as its execution context.
- Models dedicated to managing uncertainty are in most cases hybrid Petri nets that support the modelling of discrete and continuous events affecting the system, they are usually specific to a particular type of systems, giving rise to difficulties in their reuse and do not support the modelling of more complex systems.

4 Proposed model

With the emergence of complex systems, ensuring their effectiveness and efficiency is very difficult due to variations in their execution contexts and evolution in their requirements. Through self-adaptation, a system is able to cope with these contextual and environmental changes, and hence adapting to the new conditions in which it evolves. However, in the case of a quality-driven self-adaptive system, the major difficulty lies in selecting the most appropriate adaptation plans, adaptation actions and side effects that ensure and maintain the required system global qualities. Consequently, one major challenge is to deal with and to be able to exploit such uncertainty to dynamically adapt the modelled system. In this paper, we aim to combine HLPNs and PPNs to model quality-driven systems under uncertainty while considering the necessary artifacts to quantify system qualities and guide the decision-making to select the best plans with regard to quality attributes.

4.1 Overall architecture

In [5], HLPNs were adopted to formally model distributed self-adaptive systems. Although the two-layered architecture defined in [5] ensures a clear separation between the managed and the managing systems, its major drawback is that it does not consider the uncertainty concerns in the decision-making process. It also does not allow representing complex information and characteristics of a dynamic system owing to the use of ordinary Petri nets, which manipulate a single type of tokens and thus lack the necessary expressive power to model the managed system and its execution context. In addition, the managing system is defined by a collection of MAPE-K control loops, specified by means of HLPNs, in order to define a loop for each adaptation concern or quality. Such separation of concerns between system qualities may lead to negative side effects i.e. when the system tries to improve one quality, it may negatively alter other ones.

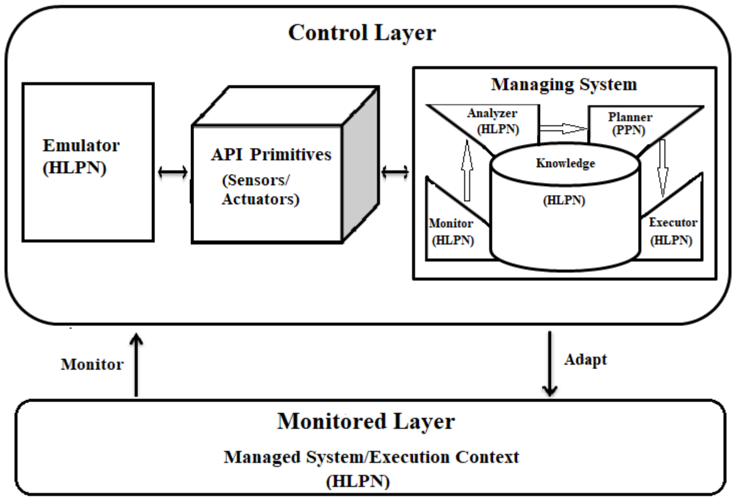


Figure 1: Approach Overview.

In our approach, we adapt and extend the model proposed in [5] to formally model quality-driven systems under uncertainty and to maintain several qualities. Our objectives through this extension are:

- to mitigate uncertainties in order to ensure the continuous satisfaction of system qualities as effectiveness, efficiency, reliability...
- to assist the decision-making process while selecting the proper adaptation plans in order to maintain the desired quality of service.
- to improve the model expressiveness and allow representing more complex information of a dynamic system; the managed system and its environment for instance.

To achieve these objectives and be able to model a quality-driven SAS, we need to firstly identify the overall qualities of the system, and then determine the system characteristics that allow their quantification as well as the contextual uncertainties that affect the system behavior. System qualities may include safety, which informally require that something bad will never happen, and efficiency, which means that the system will select the most efficient adaptation plans. We will explain these qualities in more detail in section 5. To this end, we combine HLPNs and PPNs; where HLPNs intervene in the definition of data flows through expressions and annotations; this feature will be exploited to quantify the observed qualities among the different elements of the model. On the other hand, PPNs are used to assist and improve the decision-making process and hence determine the most appropriate plans; we mainly exploit the plausibility concept of PPNs to select the most appropriate plan for an adaptation condition. We point out that, contrary to the approach adopted in [5], we model the managing system through a single control loop in order to predict and treat negative side effects of the adaptation plans on the system global qualities. Figure 1 depicts an overview of the proposed model which is an extension of the basic structure of the architecture proposed in [5] in the following directions: (1) the monitored layer is modelled by a HLPN rather than an ordinary Petri net and (2) for the control layer, the emulator is extended to support HLPNs that are also used to represent the MAPE-K elements except the planner (P) which is represented by a PPN.

This model provides a rigorous means to specify systems, ensure the continuous satisfaction of their quality and perform adaptations using the most appropriate plans.

4.2 Monitored layer

It encloses the definition of the managed system and its operating environment. We adopt HLPNs to model the monitored layer, it describes the global behavior where places model the managed system states or the context elements; transitions model actions to be performed by the system; tokens hold information about the element being modelled in that place and accept any data structure, which is the major benefit of HLPN in addition to the concept of expressions that allows quantifying system qualities.

4.3 Control layer

The control layer is generic and parameterized by the monitored layer, the quality objectives and adaptation actions. It contains (see Figure 2) the emulator, the managing system which is a combination of HLPN and PPN and the API consisting of a set of read/write primitives represented by HLPN transitions. These components provide specific roles to achieve a clear separation of concerns. To take charge of and be able to manipulate the HLPNs concepts, we have also extended and updated the API and the emulator elements; we give in what follows a brief description of the use of the emulator and the API elements (for the full description see [5]).

The emulator is used to encode the monitored layer in a specific structure modelled by an HLPN to be manipulated by firing transitions connected to places. In the emulator, each place gathers a set of elements of the managed system which are: places of the managed system together with their markings, its transitions with their guards, the input arcs with their expressions, and the output arcs with their expressions; so it has 4 places. This representation allows and facilitates the use of the API; executing the API primitives invokes the firing of the monitored layer transitions and/or adaptation of its structure. In the emulator model, the single transition move, whenever it fires, triggers the firing of a transition in the monitored layer being emulated. The emulator is connected to the managing system by means of an output arc from the transition *move* to the place *initM* of zone M in Figure 2.

The API is a set of primitives that allow reading as well as modifying the Petri nets of the monitored layer. The primitives are used inside the MAPE-K control loop to simulate the sensing and actuating actions upon the monitored layer. Each primitive is formalized by a HLPN transition (connected to specific places of the emulator) which reads or modifies the encoded Petri nets associated with the managed system in a consistent and atomic manner.

The managing system is a MAPE-K control loop specified in terms of a combination of HLPNs and PPNs sharing the same knowledge where the quantification of the observed qualities is carried out using the data flow defined by the HLPNs and the decision-making phase or the planner element is extended to assist it in selecting the best plans in the presence of uncertainty using PPNs. An overview of the managing system detailed structure is given in Figure 2 where blue places and transitions are plausible and the black transitions are the cross-zone transitions; it will be detailed in the following sub-sections.

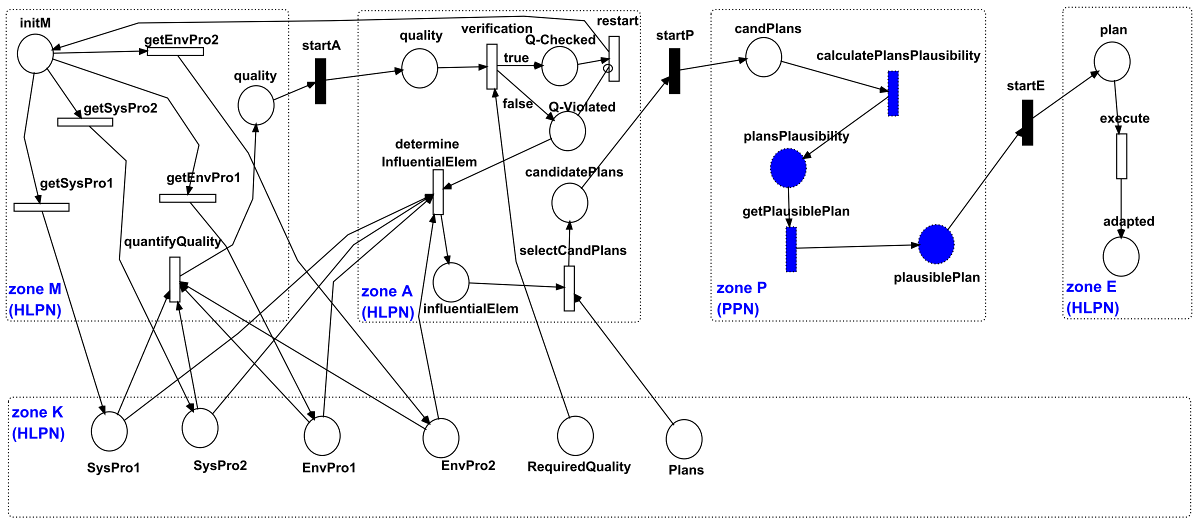


Figure 2: Detailed View of MAPE-K loop.

4.3.1 Knowledge

Since the managed system is subject to and impacted by changes in the internal/external contexts, it is inevitable to monitor them to maintain the required quality. Therefore, these contexts are represented as data hold in places of zone K (see Figure 2). Places $SysPro_1$, $SysPro_2$ define system properties, places $EnvPro_1$, $EnvPro_2$ define the current environmental properties; this data is collected and updated by the monitor element. Updates in a contextual element may provoke violation of some system quality. The contextual element is thus considered to be an influential element and will be used to determine the adaptation actions to be performed. The *Plans* place contains all possible adaptation actions that are necessary to maintain the system qualities. These actions are rules of the form condition/action.

4.3.2 Monitor

It is responsible for collecting the monitored data from the internal and external contexts of the managed system by executing transitions $getSysPro_1$ ($getSysPro_2$, ...) for the system internal context and $getEnvPro_1$ ($getEnvPro_2$, ...) for the system external context; these transitions refer to the read primitive of the API. Then, it quantifies system quality based on the characteristics of the underlying system represented by the HLPN concepts, arcs weights and place markings; the result which is a multi-set of tokens, each one representing a system property together with its actual value, is saved in the quality place.

4.3.3 Analyzer

The Analyzer element is in charge of analyzing system qualities, it represents zone A of Figure 2 and is triggered by the firing of the cross-zone transition that removes tokens from the source zone and puts them into the target zone, the *startA* cross-zone transition removes tokens of the *quality* place from zone M and puts them into zone A in the *quality* place. The transition called *verification* has as input the actual quality value and the required quality (the quality threshold). Depending on the comparison of the two values, its firing determines whether an adaptation is required or not. Whenever an adaptation is required, a step for determining the influential element is done by firing the *determineInfluentialElem* transition; since the system quality is affected by contextual elements and quality requirements. The result of this step is used as an input to select the adaptation plans that may restore the system quality. Concretely, the *selectCandPlans* transition allows selecting the candidate plans to carry out an adaptation according to the actually identified influential element, which might appear in the transition guard. At the end of the analysis phase, the analyzer triggers the planner and transmits the candidate plans to the planner using the cross-zone transition *startP*.

4.3.4 Planner

Its task is to select the proper adaptation plan from a set of candidate plans with regard to quality requirements. The Planner element is defined via a PPN which models the decision-making process and facilitates the selection of the most plausible plan to be performed in order to both restore the violated quality attribute and maintain the rest of system qualities, i.e., avoid negative side effects. The plausibility of each candidate plan is calculated, via the *calculatePlansPlausibility* transition of Figure 2, on the basis of the managed system data and the considered plan side effects on the other properties and the overall quality of the system. The most plausible plan is finally selected and transmitted via the *plausiblePlan* place to the executor element.

4.3.5 Executor

It executes the selected adaptation plan through the *execute* transition of Figure 2, representing the write primitive of the API. In fact, the managed system structure (places and transitions of the HLPN associated to the managed system) could not be changed but are rather the system characteristics, i.e., information contained in the tokens and the weight of the arcs.

The managing system cycle iterates periodically to ensure the continuous satisfaction of the system overall qualities. The various concepts introduced by the proposed model will be illustrated via the aircraft planning case study in the next section.

5 Case Study: Aircraft Arrival Planning

In order to clarify the proposed model and examine its effectiveness and efficiency, we consider the problem of aircraft planning, we first describe the problem and, identify and quantify its qualities. Then, we detail its modelling.

5.1 Problem Description

Planning and managing air operations is becoming increasingly complicated because of their congestion, but it is essential to optimize airport capacity. In order to ensure continuous traffic on the runways and to maximize the use of the airport infrastructure, a minimum level of queuing and optimal planning is required. However, the dynamics of delays and their propagation are essential elements when assessing the performance of airports [11].

Planning consists of assigning each aircraft a runway, a gateway, a pair of a terminal and a gate; consisting of the parking zone of an aircraft, to complete its departure/landing procedure. This task is usually performed as desired on the day of landing, taking into consideration two main characteristics: capacity optimization and arrival/departure safety within the airport. However, a significant number of unexpected events can occur and alter this planning and thus require its update, we can cite:

- An aircraft can arrive either later or earlier than planned. As a result, the resources allocated to it may be unavailable. The problem in this case is: should it wait until the resource is released, or should it be assigned to another resource? In this case, another question arises: which is the most efficient solution? And what are consequences on the rest of the planning, i.e., the other aircraft?
- Change of wind direction also impacts the assigned runways since an aircraft has to land with the wind direction. If the wind direction suddenly changes, would there be enough time to redo the landing plan for a set of aircraft without causing delays?
- Other events can also occur suddenly affecting the planning, such as a resource break down, or an aircraft occupying a resource longer than was estimated, adverse weather conditions, etc.

These events lead the planning system to behave in an uncertain manner; our objective throughout this paper is to manage these uncertainties by defining a model that is capable of adapting and updating the aircraft planning in the presence of a changing environment or context, but still maintaining safety and efficiency constraints. For the air operations, safety concerns maintaining a minimum separation between two consecutive aircraft, i.e., It is necessary to ensure that a certain distance always separates two consecutive aircraft according to their types. The efficiency of the air operations is achieved by minimizing delays. Efficiency consists of choosing an available resource that reduces or prevent the delays. Although these constraints are statically checked during the initial planning, they need to be verified again due to alterations and updates of the effective planning.

In this case study, we are interested in the aircraft arrival management system and its subsequent problems. The problem of arrival sequencing and scheduling at a given destination airport has been studied for several decades [13, 3, 4]. The aircraft is indeed moving in a constrained and potentially congested space. The arrival procedure consists of five phases: approach is the starting point of the procedure, i.e. the aircraft enters the airport zone; sequence consists of assigning a sequence number to the aircraft, as soon as it gets confirmation that the runway is clear, the aircraft flies to that runway; landing means that the aircraft has arrived at the runway and is landing; taxiing consists of rolling the aircraft to the terminal on its assigned track; and parking is the final phase; the aircraft has arrived at the gate where it is programmed to park.

5.2 A self-adaptive structure for the Aircraft planning system

Our proposed model is general and can be applied in several areas and on different cases. However, in order to be able to apply it to a given case study, we first project it on that case study and identify the necessary parameters of the control layer.

5.2.1 Modelling the aircraft arrival procedure

Figure 3 represents the monitored layer model of aircrafts arrival procedure. The places *Approached*, *Sequenced*, *Landed*, *Taxied* and *Parked* represent the different phases of the procedure,

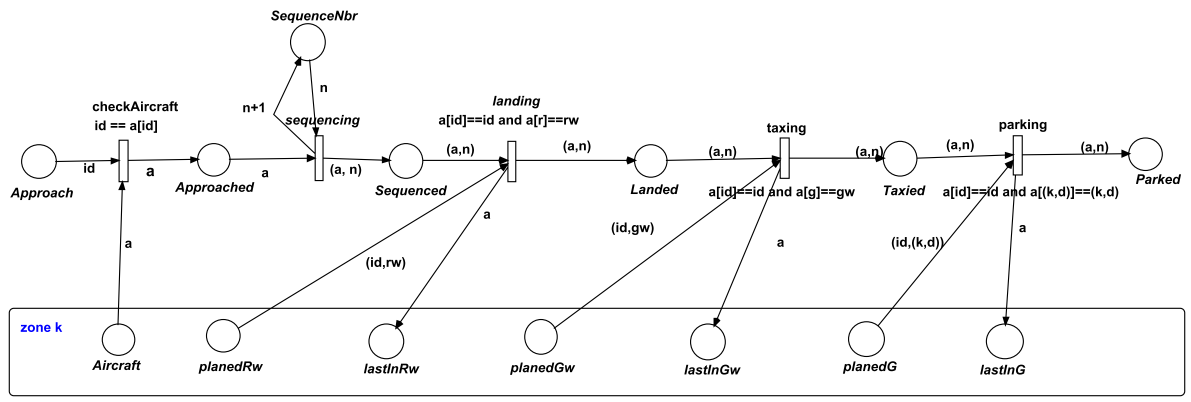


Figure 3: Formal modelling of the aircraft arrival procedure.

the presence of a token in a given place means that the aircraft has successfully finished the corresponding phase. Aircraft being at the approach phase are passed by a step of checking that they are among those planned, this is illustrated by the *checkAircraft* transition; the planning system retrieves the aircraft information and puts it in the *approached* place and assigns it a sequence number in the *sequenceNbr* place. Places *Sequenced*, *Landed* and *Taxied* constitute the inputs of the MAPE-K loop to check runways, gateways and gate availability and constraints, respectively. Places *planedRw*, *planedGw* and *planedG* represent outputs of the MAPE-K loop; tokens in these places allow the aircraft to realize its activity. Transitions represent the aircraft movement from one phase to another. In this case study, we consider the following data: an aircraft is identified by a tuple $(id, c, r, g, (k, d), ts, t, tr, tg, tp, tk, tf)$ where *id* is the aircraft identifier; c represents the aircraft category (Large (H), Medium (M), Light (L)); r, g, and (k, d) represent a runway, a gateway, and a pair of terminal and gate, respectively; *ts* is the time of the aircraft occurrence at the sequence point; t is the estimated aviation time to the landing point; *tr* is the planned landing time of the aircraft, i.e., the entrance to the planned runway; tg is the planned time to enter the planned gateway; tp is the estimated taxiing time to arrive at the parking point; *tk* is the arrival time planned to reach the gate; tf is the planned exit time of the gate. A runway is identified by (r, rs, er) where r is the runway identifier; rs is the runway state (free, occupied, or inoperative); er is the identifier of the emergency runway of the actual runway. Generally, when a runway is used, the opposite runway (in direction) is free because of wind constraint. Each gateway is characterized by (g, gs) where g is a gateway identifier; gs is the gateway state (free, occupied, or inoperative).

5.2.2 Identifying air operations qualities and constraints

The main objective of adapting the aircraft planning is to ensure both safety and efficiency requirements. Safety concerns maintaining a minimum separation distance between two consecutive aircraft and mainly avoiding collisions problems. The efficiency constraint refers to reducing delays while assigning resources to aircraft. Therefore, the aircraft arrival planning system aims to define landing plans that ensure the safety of passengers while avoiding delays through selecting the most effective ones.

Table 1: Aircraft arrival separation constraints.

Phase	Separation constraints	Description
Landing	$S(ai, ai+1) > tr(ai+1)tr(ai)$	$S(ai, ai+1)$ is the separation between an aircraft ai and its successor ai+1, and tr is the aircrafts planned landing time.
Taxing	$S(ai, ai+1) > tg(ai+1)tg(ai)$	tg is the planned time for the entrance of the gateway for ai.
Parking	$tk(ai+1) > tf(ai)$	tk is the aircraft arriving time to a gate, and tf is its exit time.

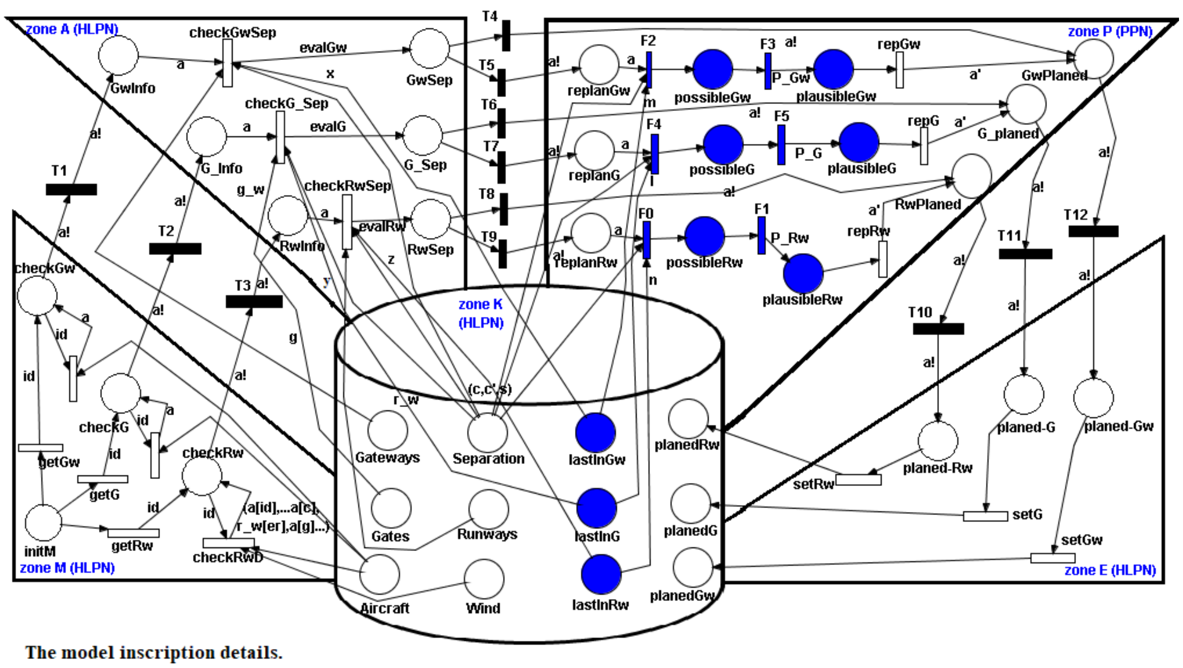
a) Securing the arrival procedure (safety)
Safety concerns maintaining a minimum separation distance between two successive aircraft at various points in the arrival procedure according to their categories [25]. In our work, it has been set at one-minute separation for cases not listed in [25]; the system administrator may still update these parameters. The separation constraint is represented by the place Separation. The analyzer is responsible for checking the arrival procedure safety at various points and then sends the results to the planner; separation constraints are resumed in Table 1 below.
b) Planning effectiveness
The effectiveness constraint is generally met by allocating new resources to the delayed/affected aircraft, but still maintaining the safety constraint and reducing the delay time. Whenever an aircraft is late, it is obvious that it will complete its procedure after the estimated time causing alterations to the successor aircraft planning and resource occupancy. In this case, it will be necessary to try to find other resources for the successor aircraft to complete its procedure on time. An aircraft may also arrive early, and hence the resources may be not yet available. In this case, new resources have to be found to avoid and reduce its waiting time. A simple measure of the planning effectiveness will depend on the safety and serviceability of the aircraft.

The wind direction metric is also considered; whenever the wind direction changes, aircraft must be reassigned to other runways. Since the opposite runway of a planned one is always unoccupied, the solution consists of switching the landings to the opposite runways while maintaining their order.

5.2.3 Maintaining aircraft planning qualities

With the aim of maintaining safety and efficiency constraints while adapting and updating the aircraft planning in the presence of a changing context, we define the model presented in Figure 4, where blue places and transitions are plausible and the black transitions (T1-T12) are the cross-zone transitions. Since the planning system is affected by several events its monitoring is required. Thus data to be monitored is: the wind direction, the airport resources states and the aircraft arrival time that allow determining the separation and the estimated overall time of the arrival the procedure.

In the knowledge zone, several places are shared and used by both the managed and managing sub-systems such as: *Aircraft* place, which contains the planned aircraft, *lastInRw*, *lastInGw* and *lastInG* places; this data is used to calculate the separation distance between two consecutive aircraft in order to ensure the arrival procedure safety and the planning effective-



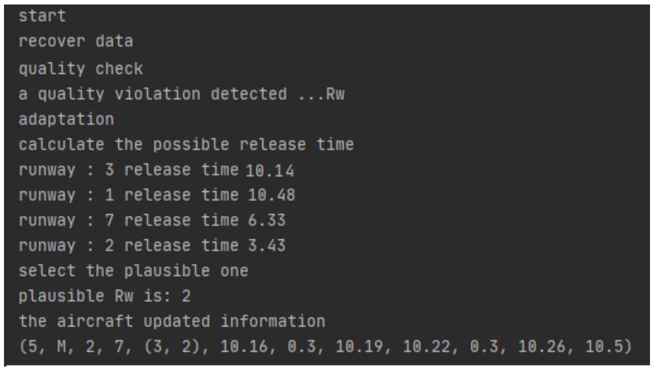
The model inscription details.

Inscription	Expression
checkRwD	a[id] == id and a[r]== ar and a[r] ==r_w[r]
checkGwSep	a[g]==x[g] and a[c]==c and x[c]==c'
checkG_Sep	a[(k,d)]==y[(k,d)] and a[c]==c and y[c]==c'
checkRwSep	a[r]==z[r] and a[c]==c and z[c]==c'
evalGw	a, ((a[tr]+a[ta])>(x[tg]+s) and a[g]==g_w[g] and g_w[gs] !=inoperative)
evalG	a, ((a[tg]+a[tp]>y[tf]) and a[(k,d)]==g[(k,d)])
evalRw	a, ((a[ts]+a[t])>(z[tr]+s) and a[r]==r_w[r] and r_w[rs] !=inoperative)
P_Gw	(a,min(f(a,m,(c,c',s))), m[g])
P_G	(a,min(f(a,l,(c,c',s))), l[(k,d)])
P_Rw	(a,min(f(a,n,(c,c',s))), n[r])

Figure 4: Maintaining safety and efficiency qualities of the Aircraft arrival procedure.

ness. For the weather conditions, we considered only wind. The actual available resources of the airport (runways, gateways and gates) are represented by the places *Runways*, *Gateways* and *Gates* respectively.

At the arrival procedure, the system uses the API read primitive *getTokens* to obtain the actual parameters and observe context changes. This primitive is represented by the transitions *getRw*, *getGw* and *getG*. Then, it checks the resource availability, if the aircraft is preparing for the landing phase, a step for checking wind direction is mandatory before landing. The analyzer firstly compares (tr(ai) + S (ai, ai+1)) with (ts(ai+1) +t(ai+1)) where ai is the last aircraft planned on the runway before the actual aircraft ai+1 and tr, S (ai, ai+1), ts and t represent the planned landing time, the required separation between the two aircraft, the time of the aircraft occurrence at the sequence point and the estimated aviation time to the landing point, respectively. The verification results are transferred to the planner where the PPN intervenes and the process of choosing a new resource is carried out. So, it calculates the possible release time of each runway by function f represented by the transition *F0* of Figure 4 where: $f = tr + s$; with s is the separation between two consecutive aircraft obtained from the place Separation and tr is



```
start
recover data
quality check
a quality violation detected ...Rw
adaptation
calculate the possible release time
runway : 3 release time 10.14
runway : 1 release time 10.48
runway : 7 release time 6.33
runway : 2 release time 3.43
select the plausible one
plausible Rw is: 2
the aircraft updated information
(5, M, 2, 7, (3, 2), 10.16, 0.3, 10.19, 10.22, 0.3, 10.26, 10.5)
```

Figure 5: Simulation results of the self-adaptive aircraft arrival planning model.

retrieved from the *lastInRw* place, which consists of a vector containing data on the last landing for each runway. After calculating the release time of all runways, they are reduced to those checking the condition: $ts + t > f$. Only one runway will be then selected; the runway with the smallest value of f; this allows selecting the best runways and thus ensures the efficiency constraint. The selection is achieved using the transition *F1* and its output arc expression, which selects the runway corresponding to the smallest f. The new runway identifier is assigned to the aircraft, and a new token is added to place *planedRw*; the *setTokens* write primitive of the API, represented by transition *setRw*, is used and executed by the executor to achieve this adaptation.
Similar operations are performed for tokens of the *Landed* place by firing transitions *F2* and *F3* to allocate a new gateway. For gate checking, tokens in the *Taxied* place are recovered and used by the plausible transitions *F4* and *F5* to allocate a new gate to the aircraft in case of the safety constraint violation.

5.3 Aircraft planning validation

The proposed model is simulated and validated using the PNemu[1] framework. To achieve such validation, we have realized some extensions and updates on the PNemu, which are missing in the original version of the source code. More precisely, we defined a new class "HLPN" to model the managed system and its environment by an HLPN; we have modified the emulator class to be able to emulate models specified as HLPN; then we have redefined some primitives of the API to capture the HLPN concepts, such as *getTokens* and *setTokens* primitives.

For brevity, we show only one scenario corresponding to the runway re-planning; we also avoid defining all models in PNemu. We assume an initial configuration of the managed system presented in Figure 3. The planned aircraft are presented in Table 2. Place *lastInRw* contains aircraft 1, 4, 7 and 9; places *lastInGw* and *lastInG* contain aircraft 9 and 7 respectively. It is assumed that aircraft 5 is planned to arrive at 9:16h, but it is arriving at 10:16h. It finished the sequencing phase and it prepares for landing, so the runway verification process is started, at this time the planned runway is occupied by aircraft 1; runway re-planning is required.

The simulation results are as follows: the actual planning of aircraft 5 violates the safety constraints due to a delay in its arrival time, i.e. the separation distance is not maintained. To

[1]PNemu has been released as open source software, available at https://github.com/SELab-unimi/pnemu.

Table 2: The actual arrival planning of the airport.

Aircraft	c	r	g	(k, d)	ts	t	tr	tg	tp	tk	tf
Aircraft 1	L	1	7	(3, 2)	10:15	2	10:18	10:21	2	10:25	10:45
Aircraft 2	M	3	5	(1, 7)	10:33	3	10:37	10:40	4	10:45	10:58
Aircraft 3	H	2	9	(6, 8)	14:15	4	14:18	14:21	1	14:25	14:45
Aircraft 4	L	7	5	(1, 7)	6:00	2	6:03	6:07	2	6:10	6:40
Aircraft 5	M	1	7	(3, 2)	9:16	3	9:19	9:22	3	9:26	9:50
Aircraft 6	H	2	9	(6, 8)	16:05	4	16:10	16:13	2	16:16	16:40
Aircraft 7	L	3	5	(3, 2)	9:40	3	9:44	9:48	1	9:50	10:20
Aircraft 8	M	7	7	(6, 8)	14:21	2	14:24	14:27	2	14:30	14:45
Aircraft 9	H	2	9	(1, 7)	3:20	2	3:23	3:27	3	3:31	4:00

restore the separation constraint, the Planner element affects a new runway to aircraft 5, runway 2 for instance. A new token is deposited in the *planedRw* place containing information (5, 2); the updated information for the aircraft is (5, 'M', 2, 7, (3, 2), 10.16, 0.3, 10.19, 10.22, 0.3, 10.26, 10.50). The adaptation details and the most plausible runway selection process are illustrated in Figure 5. If another aircraft is scheduled to land on that runway and arrives before the end of the landing operation of aircraft 5, it will be also re-planned.
Since the model is a Petri net object, it is possible to compile it and use the compiled model along with the SPOT[2] library to verify the correctness of the overall self-adaptive system with respect to design-time requirements expressed using LTL properties or another model checker.

6 Conclusion

This paper shows how to combine HLPNs and PPNs to model and analyze quality-driven self-adaptive systems evolving under uncertainty but still maintaining and guaranteeing the continuous satisfaction of an acceptable quality of service. HLPNs intervene in the definition of data flows through expression and annotation concepts, which are exploited to quantify the observed qualities among the different elements of the model. They are used for modelling the managed system and its execution context to improve the model expressiveness by representing more complex data structures of a dynamic system. PPNs are used to assist and improve the decision-making process in presence of uncertainty and hence determining the most appropriate adaptation plans through the concept of decision plausibility. An extension of the PNemu framework is also realized to take charge of HLPNs and PPNs introduced concepts as expressions and annotations. We evaluated the proposed model through the aircraft planning problem using the extended version of the PNemu framework.

As future work, we intend to explore quantitative analysis techniques to predict the impact of an adaptation plan on the overall system quality to improve the decision-making process to deal with uncertainty in the selection of both the adaptation actions and side effects and the impact of the adaptation plan on system overall qualities. We also intend to combine machine learning techniques with Petri nets to better improve the decision-making and proactivity through model training and learning.

[2][SPOT] https://spot.lrde.epita.fr

References

[1] P. Arcaini, E. Riccobene & P. Scandurra (2015): *Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation*. In Paola Inverardi & Bradley R. Schmerl, editors: *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, IEEE, pp. 13–23, doi:10.1109/SEAMS.2015.10.

[2] C. J. Baez & J. Master (2020): *Open Petri nets*. *Mathematical Structures in Computer Science* 30(3), pp. 314–341, Cambridge University Press, doi:10.1017/S0960129520000043.

[3] J. A. Bennell, M. Mesgarpour & C. N. Potts (2011): *Airport runway scheduling*. *4OR* 9(2), pp. 115–138, Springer, doi:10.1007/s10288-011-0172-x.

[4] J. A. Bennell, M. Mesgarpour & C. N. Potts (2013): *Airport runway scheduling*. *Annals of Operations Research* 204(1), pp. 249–270, Springer, doi:10.1007/s10479-012-1268-1.

[5] M. Camilli, C. Bellettini & L. Capra (2018): *A high-level petri net-based formal model of Distributed Self-adaptive Systems*. In: *ACM International Conference Proceeding Series*, ACM, pp. 40:1–40:7, doi:10.1145/3241403.3241445.

[6] A. Carl (1962): *Petri. kommunikation mit automaten*. *PhD, University of Bonn, West Germany, Technical Report RADC-TR-65–377.*

[7] M. Chiachio, J. Chiachio, D. Prescott & J. Andrews (2017): *An information theoretic approach for knowledge representation using Petri nets*. In: *FTC 2016 - Proceedings of Future Technologies Conference*, IEEE, pp. 165–172, doi:10.1109/FTC.2016.7821606.

[8] M. Chiachio, J. Chiachio, D. Prescott & J. Andrews (2018): *A new paradigm for uncertain knowledge representation by Plausible Petri nets*. *Information Sciences* 453, pp. 323–345, Elsevier, doi:10.1016/j.ins.2018.04.029.

[9] M. Chiachio, J. Chiachio, D. Prescott & J. Andrews (2019): *Plausible Petri nets as self-adaptive expert systems: A tool for infrastructure asset monitoring*. *Computer-Aided Civil and Infrastructure Engineering* 34(4), pp. 281–298, Wiley Online Library, doi:10.1111/mice.12427.

[10] A. Computing (2006): *An architectural blueprint for autonomic computing*. *IBM White Paper* 31(2006), pp. 1–6, IBM Corporation Hawthorne, NY, doi:10.1021/am900608j.

[11] A. Cook, G. Tanner, S. Cristóbal & M. Zanin (2015): *Delay propagation-new metrics, New Insights*. In: *Proceedings of the 11th USA/Europe Air Traffic Management Research and Development Seminar, ATM 2015*, EUROCONTROL/FAA , pp. 1–10, doi:10.2777/50266.

[12] D. Weyns (2017): *Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges*. *Handbook of Software Engineering*, pp. 399–443, Springer, doi:10.1007/978-3-030-00262-6_11.

[13] R. G. Dear (1978): *The dynamic scheduling of aircraft in the near terminal area*. *Transportation Research*, pp. 216–217, Elsevier, doi:10.1016/0041-1647(78)90133-8.

[14] Z. Ding, Y. Zhou & M. Zhou (2016): *Modeling Self-Adaptive Software Systems with Learning Petri Nets*. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46, IEEE, pp. 483–498, doi:10.1109/TSMC.2015.2433892.

[15] International Standard ISO/IEC 15909 (2002): *High-level Petri Nets - Concepts, Definitions and Graphical Notation*. *Final Draft International Standard ISO/IEC* 15909(4), pp. 1–43, ISO/IEC, doi:10.1007/BF02679450.

[16] R. Laddaga & P. Robertson (2004): *Self Adaptive Software: A Position Paper*. In: *Proc. of the 2004 International Workshop on Self-* Properties in Complex Information Systems*, 31, Citeseer, pp. 149–158, doi:10.1007/3-540-36554-0.

[17] J. Lee, K. F. R. Liu & W. Chiang (2003): *Modeling uncertainty reasoning with possibilistic Petri nets*. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 33(2), pp. 214–224, IEEE, doi:10.1109/TSMCB.2003.810446.

[18] R. De Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi & N. Bencomo (2013): *Software engineering for self-adaptive systems: A second research roadmap*. In: *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany*, Springer, pp. 1–32, doi:10.1007/978-3-642-35813-5_1.

[19] R. De Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi & N. Bencomo (2017): *Software engineering for self-adaptive systems: research challenges in the provision of assurances*. In: *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany*, Springer, pp. 3–30, doi:10.1007/978-3-319-74183-3_1.

[20] C. G. Looney (1988): *Fuzzy Petri Nets for Rule-Based Decisionmaking*. *IEEE Transactions on Systems, Man and Cybernetics* 18(1), pp. 178–183, IEEE, doi:10.1109/21.87067.

[21] S. Mahdavi-Hezavehi, P. Avgeriou & D. Weyns (2017): *A Classification Framework of Uncertainty in Architecture-Based Self-Adaptive Systems With Multiple Quality Requirements*. In: *Managing Trade-Offs in Adaptable Software Architectures*, Elsevier, pp. 45–78, doi:10.1016/b978-0-12-802855-1.00003-4.

[22] G. Rus, J. Chiachio & M. Chiachio (2016): *Logical inference for inverse problems*. *Inverse Problems in Science and Engineering* 24(3), pp. 448–464, Taylor & Francis, doi:10.1080/17415977.2015.1047361.

[23] S. Shevtsov, D Weyns & M Maggio (2017): *Handling New and Changing Requirements with Guarantees in Self-Adaptive Systems Using SimCA*. In: *Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017*, IEEE, pp. 12–23, doi:10.1109/SEAMS.2017.3.

[24] S. Shevtsov, D. Weyns & M. Maggio (2019): *SimCA: A control-theoretic approach to handle uncertainty in self-adaptive systems with guarantees*. *ACM Transactions on Autonomous and Adaptive Systems* 13(4), pp. 17:1–17:34, ACM New York, NY, USA, doi:10.1145/3328730.

[25] J. Skorupski & A. Florowski (2016): *Method for evaluating the landing aircraft sequence under disturbed conditions with the use of Petri nets*. *Aeronautical Journal* 120(1227), pp. 819–844, Cambridge University Press, doi:10.1017/aer.2016.32.

[26] M. Taleb-Berrouane, F. Khan & P. Amyotte (2020): *Bayesian Stochastic Petri Nets (BSPN) - A new modelling tool for dynamic safety and reliability analysis*. *Reliability Engineering and System Safety* 193, p. 106587, Elsevier, doi:10.1016/j.ress.2019.106587.

[27] J. Wang (2007): *Petri Nets for Dynamic Event-Driven System Modeling*. In: *Handbook of Dynamic System Modeling*, 1, Citeseer / Chapman and Hall/CRC, pp. 24:1–24:17, doi:10.1201/9781420010855.ch24.

[28] D. Weyns, N. Bencomo, R. Calinescu, J. Camara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J. M. Jezequel, S. Malek, R. Mirandolaand M. Mori & G. Tamburrelli (2017): *Perpetual assurances for self-adaptive systems*. In: *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany*, Springer, pp. 31–63, doi:10.1007/978-3-319-74183-3_2.

[29] Y. Zhou Z. Ding & M. Zhou (2018): *Modeling Self-Adaptive Software Systems by Fuzzy Rules and Petri Nets*. *IEEE Transactions on Fuzzy Systems* 26(2), pp. 967–984, IEEE, doi:10.1109/TFUZZ.2017.2700286.