SAFETY MONITORING FOR AUTONOMOUS SYSTEMS: INTERACTIVE ELICITATION OF SAFETY RULES

Lola Masson

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :
l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue le *21/02/2019* **par :**

Lola MASSON

SAFETY MONITORING FOR AUTONOMOUS SYSTEMS: INTERACTIVE ELICITATION OF SAFETY RULES

JURY

Simon COLLART-DUTILLEUL	RD, IFSTTAR, France	President of the Jury
Elena TROUBITSYNA	Ass. Prof., KTH, Sweden	Reviewer
Charles PECHEUR	Prof., UCL, Belgium	Reviewer
Jean-Paul BLANQUART	Eng., Airbus D&S, France	Member of the Jury
Hélène WAESELYNCK	RD, LAAS-CNRS, France	Supervisor
Jérémie GUIOCHET	MCF HDR, LAAS-CNRS, France	Supervisor

École doctorale et spécialité :
EDSYS : Informatique 4200018
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes

Acknowledgments

This work has been done at the Laboratory for Analysis and Architecture of Systems, of the French National Center for Scientific Research (LAAS-CNRS). I wish to thank Jean Arlat and Liviu Nicu, the two successive directors of the laboratory, for welcoming me. During the three years of my PhD, I worked within the Dependable Computing and Fault Tolerance team (TSF), which is lead by Mohamed Kaâniche. I thank him for the excellent work environment he has built, allowing everyone to work however it is best for them. Our work focuses on autonomous systems, and the company Sterela provided us with a very interesting case study, the robot 4mob. I thank Augustin Desfosses and Marc Laval, from Sterela, who welcomed me in their offices and worked with us on this robot. In 2017, I was lucky enough to spend three months at KTH, Sweden, in the Mechatronics division. I thank Martin Törngren, who welcomed me in the lab, and with whom I appreciated meeting while walking in the forest around the campus. I also want to thank Sofia Cassel and Lars Svensson, with whom I had very enjoyable work sessions, as well as all the other PhD students and post-docs who made my stay a great both work and personal experience. I thank Elena Troubitsyna, from KTH, Sweden, and Charles Pecheur, from UCLouvain, Belgium, for reviewing this thesis. I also thank to Simon Collart-Dutilleul, from IFSTTAR, France, for presiding on my thesis jury.

Of course, I don't forget Jérémie Guiochet and Hélène Waeselynck, from LAAS-CNRS, who supervised this work and accompanied me through the good times as well as the hard ones. Thanks also to Jean-Paul Blanquart from Airbus Defense and Space and to Kalou Cabrera, post-doc at LAAS at the time, for participating in the meetings and for their precious help and advice.

The work days of the last three years would have been different if I were not among the PhD students and post-docs of the TSF team. Thanks to those I shared an office with, providing much-needed support on the hard days and sharing laughs on the good ones. Thank to all for the never-ending card games. Special thanks to Jonathan and Christophe for the memorable nights debating, and for their open-mindness. To all, it was a pleasure hanging out and sharing ideas and thoughts with you.

Thanks to the one who is by my side every day, for his support in every circumstance. Thanks to my crew who has always been there, to my family and to all those who make me laugh and grow.

Contents

List of Figures

List of Tables

Introduction

In 2016, a shopping center in Silicon Valley acquired a surveillance robot from the company Knightscope. The robot patrolled the vicinity, collecting data through cameras and various sensors. The robot was meant to detect suspicious activity and to report such cases to security staff. On July 7 of the same year, the robot drove into a toddler, causing him to sustain a minor injury. The toddler had previously fallen on the ground and the robot was not able to detect him [Knightscope, 2016]. In 2017 in Washington D.C., the same model of this robot was patrolling office buildings when it tumbled down a flight of stairs and landed in a fountain, unintentionally destroying itself [Knightscope, 2017]. These two examples illustrate the variety of accidents that can be caused by autonomous systems. As autonomous systems become part of more application domains, from public security to agriculture, the care sector, or transportation, the number of accidents increases. Autonomous systems can cause damage to themselves and their physical environment, resulting in financial losses, but, more importantly, they can injure humans (sometimes fatally, as in the accident involving the Tesla autonomous car [NTBS, 2016]).

Ensuring safety of autonomous systems is a complex task, and has been the subject of much research in the last decades. Fault tolerance is a part of the safety domain, and is concerned with guaranteeing a safe execution despite the occurrence of faults or adverse situations. A technique that can be used for this purpose is to attach an active safety monitor to the system. A safety monitor is a mechanism that is responsible for keeping the system in a safe state, should hazardous situations occur. It can evaluate the state of the system through observation, and influence its behavior with interventions. The monitor follows a set of safety rules.

Defining the safety rules for monitors is arduous, particularly in the context of autonomous systems. As they evolve in dynamic and unstructured environments, the sources of hazard are various and complex, and a large number of rules is necessary to tolerate these hazards. At the same time, autonomous systems typically must make progress towards completing some set of tasks. The monitor needs to preserve the autonomy of the system, while ensuring that no dangerous state is reached.

Before this thesis, the *Safety MOnitoring Framework* (SMOF) has been developed at LAAS to solve this problem. It encompasses a synthesis algorithm to automatically synthesize safety rules, which prevent unsafe behaviors (safety properties) while still permitting useful ones (permissiveness properties). SMOF has been successfully applied to several case studies, but has limitations in some cases. Sometimes the synthesis cannot return a satisfying set of safety rules. When this happens, it is hard to identify why the synthesis failed, or how to construct a satisfying solution. Often, the problem stems from the failure to satisfy the permissiveness requirements: safety can be obtained only at the expense of conservative restrictions on behavior, making the system useless for its purposes.

In this thesis, we propose an interactive method for the elicitation of safety rules in case SMOF's synthesis fails to return a satisfying solution. To assist the user in

these cases, three new features are introduced and developed. The first one addresses the diagnosis of why the rules fail to fulfill a permissiveness requirement. The second one allows the tuning of the permissiveness requirements based on a set of essential functionalities to maintain. The third one suggests candidate safety interventions to inject into the synthesis process.

In Chapter 1, we present the context of this work. We give a high level view of the different means for ensuring the safety of autonomous systems, before focusing on the definition of safety monitors. We also explain the main concepts of SMOF.

Chapter 2 presents the results of applying SMOF to a new case study—a maintenance robot from the company Sterela. We also revisit the results of a previous case study involving a manufacturing robot from the company KUKA. In light of these two case studies, we discuss the main limitations of SMOF and the manual solutions that were implemented for them. We deduce tracks for improvements, and identify the needs for our contributions.

In Chapter 3, we come back to the problem identified in Chapter 2 where no satisfying solution can be synthesized and discuss it in detail. From there, we present our contributions, and how they fit together for solving this issue.

Chapters 4, 5 and 6 present the detail of our contributions. In Chapter 4, we detail a diagnosis tool that can help the user identify why the synthesis failed to return a satisfying solution. In Chapter 5, we define a template for better adapting the synthesis to the system's expected tasks. Finally, in Chapter 6, we present an interactive method for defining new interventions.

We conclude with a summary of the contributions and present some perspectives and thoughts for future work.

List of publications:

- Lars Svensson, Lola Masson, Naveen Mohan, Erik Ward, Anna Pernestal Brenden, et al.. Safe Stop Trajectory Planning for Highly Automated Vehicles: An Optimal Control Problem Formulation. 2018 IEEE Intelligent Vehicles Symposium (IV), Changshu, China, Jun. 2018.

- Mathilde Machin, Jérémie Guiochet, Hélène Waeselynck, Jean-Paul Blanquart, Matthieu Roy, Lola Masson. SMOF - A Safety MOnitoring Framework for Autonomous Systems. In IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 48, no 5, pp. 702-715, May 2018. IEEE.

- Lola Masson, Jérémie Guiochet, Hélène Waeselynck, Kalou Cabrera, Sofia Cassel, Martin Törngren. Tuning permissiveness of active safety monitors for autonomous systems. In Proceedings of the NASA Formal Methods, Newport News, United States, Apr. 2018.

- Lola Masson, Jérémie Guiochet, Hélène Waeselynck, Augustin Desfosses, Marc Laval. Synthesis of safety rules for active monitoring: application to an airport

light measurement robot. In Proceedings of the IEEE International Conference on Robotic Computing, Taichung, Taiwan, Apr. 2017.

- Lola Masson, Jérémie Guiochet, Hélène Waeselynck. Case Study Report : Safety rules synthesis for an autonomous robot. In Proceedings of the Fast abstracts at International Conference on Computer Safety, Reliability, and Security (SAFECOMP), Trondheim, Norway, Sep. 2016.

Chapter 1

Ensuring the Dependability of Autonomous Systems: A Focus on Monitoring

Contents

Autonomous systems bring new challenges to the field of dependability. They have complex non-deterministic decision mechanisms, evolve in unstructured environments, and may be brought to interact with humans or other systems to carry out their tasks. They are thus subject to new types of faults. The classical dependability techniques need to be adapted to encompass these new constraints. This chapter starts by presenting a high-level view of the state of the art on the dependability means, applied to autonomous systems (Section 1.1). It then gradually focuses the discussion on the safety monitoring approach—SMOF—that we use and extend in our work.

In Section 1.1, active safety monitors are introduced in relation to one of the dependability means, namely fault tolerance. Such monitors are responsible for ensuring safety of the system despite its faults: they observe the operation of the system, and are able to trigger corrective actions to prevent hazardous situations from occurring, according to some safety rules. Section 1.1 discusses how monitors can be introduced at various

architectural levels to serve different purposes. They can also be separated from the control architecture, to act as the ultimate barrier against catastrophic failures.

Section 1.2 provides more details on active monitoring, with an emphasis on the core techniques. Two broad classes of techniques are identified, coming from distinct communities. The property enforcement mechanisms from the runtime verification community manipulate traces of events to ensure their correct sequencing and timing. In contrast, the reactive monitors from the safety-critical control system community have a focus on state invariants: the values of variables are monitored and actions are taken to keep these values inside a safety envelope.

Section 1.3 provides an overview of SMOF, the safety monitoring framework developed at LAAS. This framework targets reactive monitors that are independent from the control architecture. SMOF has been conceived to solve the problem of safety rules specification in the context of autonomous systems. It provides a synthesis algorithm based on model-checking, to synthesize safety rules from a model of the available observations and actions, along with the safety invariants to hold. The invariants are extracted from a hazard analysis.

Section 1.4 concludes the discussion.

1.1 Concepts and Techniques for Safe Autonomous Systems

[Avizienis et al., 2004] propose four means to classify techniques for ensuring dependability of systems:

Fault prevention: preventing the occurrence or introduction of faults, through rigorous engineering methods, and the use of dedicated tools (Section 1.1.1);

Fault removal: reducing the number and severity of the faults, using verification techniques (Section 1.1.2);

Fault forecasting: estimating the number of faults and their consequences, mostly using risk analysis methods (Section 1.1.3);

Fault tolerance: avoiding service failure despite the presence of faults remaining after the use of the previous methods, or introduced during the operational life of the system (Section 1.1.4).

1.1.1 Fault Prevention

Fault prevention aims at preventing the occurrence or introduction of faults in a system and is part of general engineering techniques for system development. Autonomous systems are very complex and the diversity of their tasks forces developers to deal with heterogeneous components when developing their software. In [Blanquart et al., 2004], the authors draw recommendations for the design of dependable autonomous systems:

they must have a structured architecture, some standardized components and interfaces, and automatically-generated code (as much as possible). These recommendations are gathered in what is called *model-based development*. To comply with the first requirement, several architectural frameworks for autonomous systems exist. The most widely used is the hierarchical architecture (see LAAS [Alami et al., 1998], IDEA [Muscettola et al., 2002], CLARAty [Volpe et al., 2001]). The idea is to split the decision making into several levels, each one using a different level of abstraction. Classically, the hierarchical architecture is divided in three layers:

• **The decisional layer** uses an abstract representation of the system and its environment to compute plans to achieve some objectives. These objectives can be sent to the system by another system or a human operator. This layer does not guarantee a real time execution.

• **The executive layer** converts plans computed by the decisional layer into sequences of tasks that will be performed by the functional layer. This layer is sometimes merged with one of the other layers.

• **The functional layer** executes the tasks. This layer is responsible for commanding the actuators, retrieving the sensors' data, computing trajectories, etc. It is often composed of components that can interact but do not have a global view of the system.

Each layer sends the results of the different tasks' executions, and some potential errors that could not be handled, to the higher level. Some hybrid versions of the hierarchical architecture exist, making direct communications available between the functional and decisional layer, or combining layers together.

The recommendations for standardized components, architectures and interfaces [Blanquart et al., 2004, Bruyninckx et al., 2013] can be addressed by the use of middleware like ROS [Quigley et al., 2009], OROCOS [Bruyninckx, 2001], and by component-based frameworks like GenoM [Mallet et al., 2010] or Mauve [Lesire et al., 2012]. The component-based frameworks can also provide code generation facilities, thereby reducing the probability of introducing development faults. In some cases, these frameworks are also linked to formal methods (mathematically-based techniques used to specify, develop or verify software systems). GenoM automatically synthesize formal models that can be used for checking temporal properties (with FIACRE/Tina [Fiacre, 2018, Tina, 2018, Foughali et al., 2018], or with BIP [Basu et al., 2006]). Mauve provides facilities for analyzing execution traces.

Controller synthesis: Supervisory control synthesis (SCT) [Ramadge and Wonham, 1987] also contributes to fault prevention. It is a technique that aims at automatically generating a suitable controller for a software system. From a model of the system as an automaton, and a set of properties to satisfy, a synthesis mechanism is used to generate the controller (see Figure 1.1). The controller restricts the capabilities of the system, by triggering controllable events, according to the specified properties. In [Shoaei et al., 2010], a controller is synthesized for manufacturing robots in a shared environment. The authors propose to generate non-collision properties from simulation and formal models of the robots operations, and to integrate these in the synthesis, resulting in collision-free manufacturing units. [Lopes et al., 2016]

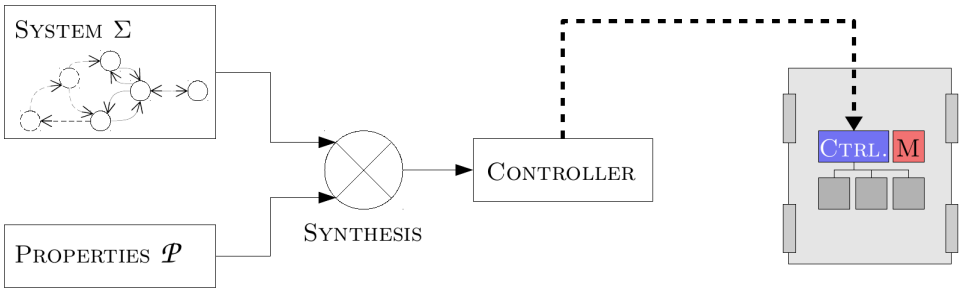

Figure 1.1: Concept of supervisory control synthesis

successfully apply controller synthesis for swarm robotics — large groups of robots interacting with each other to collectively perform tasks. In [Ames et al., 2015], the authors use controller synthesis for a walking robot. The synthesized controller shows improved performance compared to the manually-implemented controller initially used. Robotics seems to be a promising application field for controller synthesis.

1.1.2 Fault Removal

Fault removal aims at removing faults from the considered system. It can occur during the development phase, where faults are found and corrective actions taken before deployment of the system; or during its use, where faults that have occurred are removed (corrective maintenance) and actions are taken to avoid the occurrence of new faults (predictive maintenance). In this section, we will focus mainly on the fault removal techniques that can be used during the development phase. Removing faults from a system is done in three steps. The verification step checks that the system conforms to some properties. If the verification reveals a property violation, the diagnosis step identifies what fault led to the the property violation and the correction step brings the necessary changes to the system to remove the fault. If the diagnosis and correction steps are very dependent on the considered system, the verification techniques are more generic — we will detail the most common ones in the following. The verification techniques can be classified into two categories: static verification, which does not need to execute the system, and dynamic verification which does. Autonomous systems are more challenging to verify than classical control systems [Pecheur, 2000]. A full model of the system is usually not available or is hard to build, the number of scenarios to be analyzed is infinite, and the software is non-deterministic (mostly due to concurrency). However, formal methods are widely used with success for the verification of autonomous systems, and the user can refer to the extensive survey in [Luckcuck et al., 2018] for more references.

Static verification aims at mathematically proving that a program complies with some requirements. It guarantees that all executions are correct according to the con-

sidered requirements. It includes techniques like theorem proving or model-checking. These techniques use a model of the system.

Model-checking exhaustively searches the state space of a model to determine if some property is verified. To define properties, temporal logics like Computation Tree Logic (CTL) are widely used. In [Dixon et al., 2014], the authors propose to verify the control rules of a robotic assistant. The rules —expressed in Linear Temporal Logic (LTL)— defining the allowed behaviors of the robots are manually translated (and later on automatically translated by their tool CRutoN [Gainer et al., 2017]) into suitable inputs for a model checker. The model checker then checks for satisfiability of some properties (priority between rules, interruptibility). Stochastic model-checking is an extension of model-checking to deal with uncertainties that are inherent to autonomous systems [Kwiatkowska et al., 2007]. It evaluates the probability that a property is satisfied within a scope. Examples of such approaches for autonomous robots can be found in [O'Brien et al., 2014] or [Pathak et al., 2013].

Verifying the decision algorithms used in the decisional layer is challenging. In [Fisher et al., 2013], model-checking is used to verify all the possible decisions made by the decision mechanism of autonomous systems. The authors can then ensure that the system will never intentionally make the decision to move towards an unsafe situation. However, an unsafe situation could still occur, for instance due to an unexpected change in the environment (it is impossible to have a complete model of the environment).

In [Desai et al., 2017], the authors propose to use model-checking combined with runtime monitors. For the model-checking of the robot software, one has to make assumptions about the low-level controller and the environment. The addition of an on-line monitor allows one to check that these assumptions hold at runtime. A similar approach is explored in [Aniculaesei et al., 2016], where the model checker UPPAAL is used at design time to verify that no collision can occur. A monitor is used at runtime to verify whether the conditions used for the model checking still hold.

The exhaustive aspect of model checking makes the approach appealing, but it has some limitations. Complex systems can have extremely large (infinite) state spaces, making them hard to completely explore (state explosion issue). Also, the modeling step is very error-prone, and it is hard to obtain a model that is both accurate enough to represent the system and abstract enough to limit the state explosion. A deep understanding of both the application-to-verify and the model-checker used is needed.

In *theorem proving*, a formal proof of the correctness of a system is produced, through logical inferences. In [Täubig et al., 2012], an obstacle avoidance algorithm for static obstacles is proved correct, using the theorem prover Isabelle. This process produced a robust argumentation that allowed the authors to get a SIL3 of IEC 61505 certification for the presented method and implementation. In [Huber et al., 2017], the authors formalize traffic rules for vehicle overtaking. They use the help of the Isabelle prover to achieve a high level trustworthiness of the formalization. Though theorem proving is an efficient verification technique, it has not been extensively investigated in the field of robotic systems. The tools are generally difficult to master and require a higher expertise compared to other approaches.

Dynamic verification consists in executing the system or a model of the system in a specific context in order to analyze its behavior. A set of inputs (called a *test case*) is given to the system. For symbolic inputs, we talk about *symbolic execution*. For valued inputs, we talk about *test*. The outputs are then analyzed, and a pass/fail verdict is emitted by a procedure called the *test oracle*. Defining the test oracle for autonomous systems is not trivial: they are nondeterministic and their tasks are complex and various. To overcome this issue, [Horányi et al., 2013] propose to automatically generate test cases from UML scenarios: the traces are evaluated against the requirements expressed by the scenarios. Injecting faults is another way to define test cases, and is explored in [Powell et al., 2012] for autonomous systems. For testing autonomous systems on a mission level, field testing is used. This is expensive and potentially dangerous — as testing robots in a real environment can damage the robot or its environment; and only provides a limited coverage —as only a limited number of situations can be tested in a limited time. For these reasons, the development of simulators is a major improvement. However, the result of tests in simulation are valid only assuming that the simulation is representative of the real behavior. Several simulation environments exist and the reader can refer to [Cook et al., 2014] for a comparison. Drone collision-avoidance is tested in simulation in [Zou et al., 2014]. Evolutionary search is used to generate challenging test cases, and proves to be efficient in identifying remaining hazards. Generating the simulated environment is challenging. The environment needs to closely represent the world in which the robot will evolve once deployed. However, as shown in [Sotiropoulos et al., 2017], it is possible to identify software bugs in simulation in a simpler environment than the one into which the robot will be introduced. The same authors study the difficulty of the generated map in [Sotiropoulos et al., 2016], and how to make it vary by tuning parameters like obstruction and smoothness. Randomness can be introduced in the generation of the simulated world, as in [Alexander and Arnold, 2013] where the authors test the control software of a robot on situations as varied as possible. In [Abdessalem et al., 2018], evolutionary algorithms are used to guide the generation of scenarios to test for autonomous vehicles.

1.1.3 Fault Forecasting

The idea of fault forecasting is to evaluate what faults may occur and what would be their consequences. It is mainly done by performing risk analyses that "systematic[ally] use [the] available information to identify hazards and to estimate the risk" [ISO/IEC Guide 51, 2014]. The risk is evaluated using qualitative levels such as "low" or "high". Different risk analysis techniques can be applied at different stages of the development, to different components and abstraction levels.

The risk analysis techniques are classified into two categories: *bottom-up*, that analyze the effect of faults for known deviations (e.g., Failure Modes Effects and Criticality Analysis (FMECA), Hazard Operability (HAZOP)) or *top-down*, that identify which faults lead to an undesired event (e.g., Fault Tree Analysis (FTA)). These methods are commonly used for autonomous systems [Dhillon and Fashandi, 1997], but they are facing some challenges. Due to the complexity and non-determinism of the decision

mechanism, it is hard to predict the consequences of faults, or even to trace back causes of known consequences. Also, it is hard to predict the probability of occurrence of a specific event in an unstructured environment. In [Crestani et al., 2015], the authors apply the FMECA method to identify the risks of autonomous systems. They use these results to implement some fault tolerance mechanisms.

The HAZOP technique associates a guide word like "more" or "late" to each element of the analyzed system, to help the user identify hazards associated with these deviations. It has been used on a therapeutic robot for children in [Böhm and Gruber, 2010] where the system is decomposed into components and functions to be thoroughly analyzed. A variant dedicated to software called HAZOP-SHARD (Software Hazard Analysis and Resolution in Design) is used in association with a specific list of environmental risks in the context of autonomous systems [Woodman et al., 2012]. In [Troubitsyna and Vistbakka, 2018], HAZOP is combined with the analyses of data flows to identify safety and security requirements. These requirements are then formalized in Event-B to analyze the dependencies between the architectures' components and the safety and security mechanisms. In [Guiochet, 2016], the HAZOP technique is adapted to the analysis of UML diagrams. The user systematically analyzes all the components of the diagrams with the help of guide words. This method is one of the few focusing on human-robot interaction.

Also focusing explicitly on human-robot interaction, SAFER-HRC is a semi-automatic risk analysis for Human Robot Collaboration (HRC) applications [Askarpour et al., 2016]. A team of experts builds a model of the studied system and the possible human-robot interactions, and identifies what hazards may occur from the list presented in [ISO 12100, 2013]. This model is then analyzed to identify— knowing the dynamics of the systems— what interaction accidents may occur; some corrective actions can then be taken. [Stringfellow et al., 2010] introduce the STPA (Systems Theoretic Process Analysis) method that considers a model of the potential faults leading to accidents. From it, they identify which erroneous control commands could lead to an accident. This method has been applied to a medical robot in [Alemzadeh et al., 2015]. In the method ESHA (Environmental Survey Hazard Analysis) [Dogramadzi et al., 2014], new guide words are used to analyze the interactions with the environment. They specifically take into account unexpected interactions (not planned on in the mission), as they are the ones causing the most damage, as the authors claim. Several methods can also be used in combination. In [Alexander et al., 2009], Energy Trace and Barrier Analysis (ETBA) is used as a preliminary analysis. In this method, the release of energy is analyzed— an accident occurs when an unexpected or excessive amount of energy is released. The Functional Failure Analysis (FFA) method is also used, as well as HAZOP. The results of these hazard analyses are then used in an FTA to identify their causes. Though these methods are not exhaustive, they are systematic and cover a large amount of hazards. Among the identified faults, some cannot be removed: they have to be tolerated.

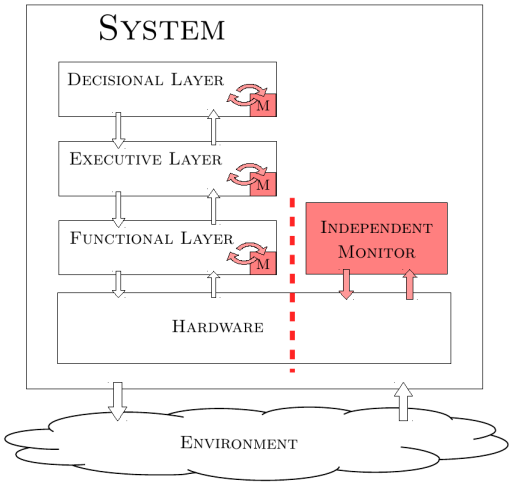

Figure 1.2: Monitors in a hierarchical architecture

1.1.4 Fault Tolerance

Fault tolerance aims at providing a correct service despite the occurrence of faults. This is essential, as no complex system can be considered fault-free. This is particularly true of autonomous systems, which include complex decisional functions and may face adverse and unspecified situations. To mitigate hazards during the operational life of the system, some mechanical (a specific shape, a soft cover, etc) or hardware (bumpers, etc) ways exist but we will not detail them here and only consider software mechanisms. Fault tolerance is achieved in two steps: fault detection and recovery. This implies that the mechanism has ways to observe faulty behaviors and ways to trigger corrective actions. Several approaches for fault tolerance exist. For instance, redundant sensors can detect inconsistent values, or a backup software component can be used if the primary one crashes. Another approach is to design a specific component that can be either integrated into the architecture or independent of it, and this is what we will be interested in. We will refer to this component as a *monitor*. A monitor is only responsible for safety: it checks whether some properties are violated or not, and potentially triggers relevant corrective actions. Some monitors are designed specifically to tolerate the fault of one or several layers in a hierarchical architecture (see Section 1.1.1). They interact with a single layer, as represented in Figure 1.2.

In the functional layer, the monitor is mostly responsible for sensors and actuators errors. In [Crestani et al., 2015] a monitor is attached to each hardware and software component of the functional layer. Similarly, in [Zaman et al., 2013], such monitors are integrated in a ROS architecture. These monitors are able to detect errors such as sensor defects or timeout issues. They are very similar to classical fault tolerance approaches such as redundancy or watchdogs.

In the executive layer, the monitor will evaluate the results of executions from the functional layer components and the commands coming from the decisional layer. It can manage the tasks execution when resources are missing, trigger reconfiguration if needed or filter requests. In [Durand et al., 2010], a monitor can adapt the autonomy level of the robot in order to tolerate faults in the functional layer. In [van Nunen et al., 2016], the authors propose to design a fault tolerance unit, attached to the controller, that is able to make decisions in order to avoid collisions for trucks following each other (platooning). The component selects the safe state to evolve towards, and triggers a safety action to reach it. In [Py and Ingrand, 2004], the monitor is integrated between the functional layer and the decisional layer, replacing the traditional executive layer. It can block erroneous requests from the decisional layer and interrupt the execution of functions. The same approach with different technologies is explored in [Bensalem et al., 2009], where a monitor is automatically synthesized, following an idea close to controller synthesis. [Huang et al., 2014] propose a single component called RVMaster, integrated in ROS, that supervises the others, verifying some safety properties. Even though this component is not strictly speaking integrated in a hierarchical architecture, it can be assimilated into an executive layer as it is supervising the execution of the functional components.

In the decisional layer, the monitor usually checks that the plan satisfies some safety constraints or that the planner isn't faulty. In [Ertle et al., 2010], a model of the environmental hazards learned prior to deployment and updated on-line is used to evaluate the generated plan with regard to safety principles. In [Gspandl et al., 2012] the decisional layer is able to tolerate faults from the functional layer by comparing the current state to a correct state. Considering faults of the planners themselves, [Lussier et al., 2007] proposes to tolerate them using redundant diversified planners. Similarly, [Ing-Ray Chen, 1997] proposes to tolerate intrinsic planning faults (failure of the planning algorithm) using a parallel architecture.

Multi-layer mechanisms: [Gribov and Voos, 2014] propose a multi-layer architecture, that associates to every hazard a software component treating them. This approach is useful as it can be integrated into any existing ROS architecture, and helps ensuring safety when off-the-shelf components are used. However, the safety mechanisms are spread at different levels, which is a disadvantage as it does not allow the treatment of safety of the system as a whole. In [Vistbakka et al., 2018], a multi-layered approach is used for the safe navigation of a swarm of drones. A high level decision center is in charge of the safe navigation of the drones among known hazards. It can change the swarm configuration or generate new routes for the drones. Additionally, at a low level, each drone has its own local collision avoidance mechanism, that computes "reflex movements" to avoid or mitigate collisions with unforeseen obstacles.

Independent monitors: Another approach for monitoring safe behavior of an autonomous system is to design an independent component that is separated from the control channel. This idea is represented by the large gray box in Figure 1.2. This component needs to be independent so that potential errors in the main control channel do not propagate to it. It also needs specific ways to observe the system's state and

to intervenes should a deviation be perceived. This type of component is also referred to in the literature as a safety bag [Klein, 1991], safety manager [Pace et al., 2000], autonomous safety system [Roderick et al., 2004], guardian agent [Fox and Das, 2000], or emergency layer [Haddadin et al., 2011]. In [Tomatis et al., 2003], an autonomous tour-guide was successfully deployed for five months using, among other safety mechanisms, an independent monitor that is implemented on its own hardware component. It only has limited means of observation and intervention, but will not be affected by any fault of the main software part. In [Woodman et al., 2012], a monitor is able to filter commands to actuators or stop the robot in case of a safety rule violation. The monitor also intervenes if it detects a strong uncertainty in the observations. It is not independent from the controller in terms of observation means: they both share the same view of the system state, therefore potentially the same errors. In [Casimiro et al., 2014], the authors propose to implement a *safety kernel* on a dedicated hardware board. The safety kernel detects the violation of a set of safety rules defined in design time. It detects the operational context (e.g., sensor defect) and can adjust accordingly the *level of service* of the appropriate functions. For instance, if the distance sensor is faulty, the safety kernel can adjust the safety distances margin to ensure that no collision can occur. In [Machin et al., 2018], a safety monitor implemented independently detects when a warning threshold is crossed, and can trigger appropriate safety interventions to prevent the system from evolving towards a hazardous state.

As can be seen, monitors play useful roles at the various layers of the architecture. Generally speaking, the monitoring integrated into the control channel offers more powerful detection and recovery mechanisms than an independent monitor. The latter category of monitor has more limited observation and intervention means, but can act as the ultimate barrier against faults when all other mechanisms fail in their protection role. It is typically kept simpler and assigned a high integrity level. In the next section, we discuss the existing monitoring techniques and key options underlying the design of monitors for autonomous systems.

1.2 Monitoring Techniques

Monitoring involves some verification at runtime to detect a potentially dangerous behavior. This verification concern allows for potential cross-fertilization between fault tolerance and fault removal techniques. Indeed, *runtime verification* (RV) has emerged as a fruitful field of research in recent years, with close relationships with model checking and testing. We first describe the main concepts of RV, the similarities and differences compared to the other verification techniques, and comment on the type of monitors considered by the RV community (Section 1.2.1). We then discuss other types of monitors considered for autonomous systems, which accommodate a more reactive view (Section 1.2.2).

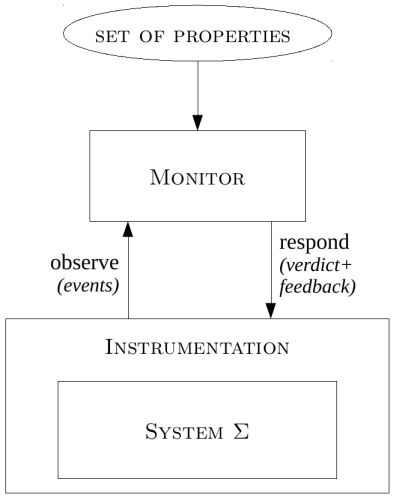

Figure 1.3: Typical process of runtime verification (adapted from [Falcone et al., 2013])

1.2.1 Runtime Verification

We synthesize here the views of [Leucker and Schallhart, 2009], [Falcone et al., 2012], [Falcone et al., 2013] and [Delgado et al., 2004] that the reader may refer to for further information. [Falcone et al., 2013] defines Runtime Verification as "a dynamic analysis method aiming at checking whether a run of the system under scrutiny satisfies a given correctness property". The idea is to check a set of properties against a system's execution.

Typically, the RV framework is described as in Figure 1.3.

First, a monitor is generated from a property, typically specified with an automata-based or logic-based formalism. This step is referred to as *monitor synthesis*. The monitor is a decision procedure for a property. It observes the execution of the system through *events*, and generates a *verdict* on the satisfaction of the property. The verdict produced by the monitor could be two-valued (the property is satisfied or not), but can also have additional values allowing a finer evaluation of the property. The verdict generated can be accompanied with a *feedback*, which gives additional information to the system, for instance in order to trigger corrective actions. Classically, recovery is referred to as *runtime enforcement*. The runtime enforcement mechanism detects (in anticipation) the violation of a property. It can then delay or block some events, to avoid the violation. The authors of [Ligatti et al., 2009] propose to formalize and analyze the types of properties that can be enforced by a runtime monitor.

Second, the system is instrumented to be able to generate the appropriate events. This step is highly dependent on the system itself and the technologies used.

Third, the system is executed and its traces analyzed. The analysis can occur during the system execution (*online* monitoring), or after the execution if the events have been stored (*offline* monitoring). When the monitor in included in the system's code itself,

it is called *inline*. This is conceptually very close to the monitors integrated in the architecture presented in Section 1.1.4. When the monitor is its own entity, it is called *outline*. This is similar to the independent monitors presented in Section 1.1.4.

RV receives growing attention in the field of autonomous systems (see [Luckcuck et al., 2018, Pettersson, 2005]). For instance, in [Lotz et al., 2011], RV is used for monitoring robotics components, both during the development process for debugging, as well as at runtime for fault and error detection and handling. An interesting aspect is that direct code instrumentation is seldom desirable for autonomous systems. The source code of such systems may not be available, and the modification of the code may affect timing and correctness properties — which can also invalidate a former certification, resulting in non-negligible additional costs. [Kane et al., 2015] propose to build a passive RV monitor that gets the appropriate events from passive observation of the communication buses. They successfully apply this approach to an autonomous research vehicle. Similarly, in [Pike et al., 2012] the authors propose to delegate fault tolerance to a monitor external to the system itself. The monitor is passive with regard to the system's execution.

Runtime verification and model checking: RV has its source in model checking. They both aim at verifying that executions of a (model of a) system satisfy a property. However, model checking considers *all* possible executions of the system, whereas runtime verification only analyzes a subset of them. In model checking, the available resources are less of an issue than for monitoring, especially online monitoring. For instance, backward search is commonly used in model checking and is not desirable in monitoring as it would imply storing the whole execution trace. Model checking may suffer from state explosion, as the systems (and their models) become larger. In runtime verification, monitors check one run at a time, thereby avoiding this issue. RV can be applied to a "black box" system, and does not need a model of the system (that can be hard to build). Model checking requires a model of the system to be built so that all the possible executions can be checked. However, when a finite model can be built model checking and RV can be used together, where monitors check that the assumptions used for the model checker hold at runtime, as in [Desai et al., 2017].

Runtime verification and testing: RV is very similar to what is called *passive testing*. Passive testing aims at testing systems for which classical (active) testing is hard to implement. The system is executed, and the outputs or traces of execution are analyzed to detect faults, in order to remove them. RV is interested with ensuring the proper functioning of systems that are already deployed. If RV and passive testing have different goals, they share similar techniques. In both cases, the system has to be instrumented to allow for the observation of the interesting sequences of events. The formalisms used for these two techniques (automata-based, etc) are also similar. For instance, in [Cavalli et al., 2003], the authors propose to use an extended form of a Finite State Machine as a specification for passive testing. An extensive analysis of the relations between RV and testing is beyond the scope of this work, but the reader can refer to [Falzon and Pace, 2013] for some work on extracting runtime monitors from a

testing specification, or to [Goldberg et al., 2005], where the authors use RV to build test oracles for the autonomy software of a planetary rover.

For the RV community, the objects of interest are events. The verified properties define the correct sequences of event. In the case of a property violation, the enforcement mechanism can insert, delete or delay events in order to obtain a correct trace. As pointed out in [Bloem et al., 2015], this approach is not necessarily appropriate for reactive systems, for which reactive monitoring may be privileged. The enforcement then consists in preventing certain combinations of variable values that characterize a hazardous state.

1.2.2 Reactive Monitors for Autonomous Systems

Autonomous systems are reactive systems. They continuously observe their environment through sensors, can make decisions, and command the actuators to trigger changes. In this context, the *rules*, i.e., the way the monitor should behave in certain conditions, are most of the time a list of if-then-else conditions, e.g., if the platform goes too fast, then brake. We call a *strategy* the set of rules the monitor follows. It encompasses what risk is covered by each rule, what recovery action must be taken and what the condition is to trigger it. Defining these rules needs a well structured method, as they can be complex and numerous, due to the complexity inherent to autonomous systems. The framework <observation> then <action> brings us to consider several aspects: what is observed, what actions are possible, how to associate actions to specific observations (synthesis), and how to express the properties verified by the strategy.

Observations: They can be directly extracted from the sensor values, or can result from a more sophisticated calculation within the monitor. The authors of [Feth et al., 2017] propose a framework to design a safety supervisor. They propose to use what they call the *safety world model* to identify the situation in which the system is, and evaluate its risk. It relies on an internal representation of the environment, a prediction of the potential evolution of the situation and an evaluation of the risk of this situation. In [Machin et al., 2018], however, the authors simply compare sensor values to predefined thresholds (possibly with a small calculation to extract the relevant information) to evaluate how close to the hazardous situation the system is.

A sophisticated analysis of the situation makes it possible to better anticipate the danger, and to trigger complex recovery actions. However, a simpler data analysis is consistent with the concept of an independent safety monitor that acts as the last barrier for safety.

Monitor actions: Several ways exist to recover from a failure, i.e., to bring the system back to a safe state. In the case of autonomous systems, the recovery is ideally such that the system can keep operating normally, or at least perform a subset of its tasks. The recovery actions may need to be complex, while remaining reliable. However, in most cases the main available recovery action is to shut down the robot, which is very conservative. Shutting down the robot might also not be the best action for safety. Indeed,

in [Malm et al., 2010] the authors show that most accidents resulting from human-robot interactions come from crushing a person against an object. In this case, shutting down the root is not desirable as the person will stay stuck, resulting in a worse injury.

In [Arora et al., 2015], the authors present a set of emergency trajectories computed off-line, that can be used on-line for ensuring the safety of a helicopter. Instead of computing a trajectory to a safe state during operation, which is likely infeasible, the safety system can pick one from the library. The problem remains that reaching a safe state following these trajectories must be guaranteed, i.e., the component responsible for the execution of a trajectory must be certified. However, this avoids certifying the whole planner. This approach is also less conservative than using the traditional emergency stop. Similarly, in [Feth et al., 2017] the supervisor can select from among a library of strategies one that will lead it to a less critical state. The authors note that if the supervisor aims at remaining simple, an overly-simple one may not be satisfying with regard to autonomy. Therefore, they propose some higher layers of supervision that may be able to trigger more complex behaviors.

An interactive approach is explored in [Durand et al., 2010] where the authors propose to reduce the autonomy level of the system. A human operator can take over the tasks that can no longer be automated because of a failure.

In [Machin et al., 2018], the interventions are expressed as preconditions and effects on the state of the system. They mostly rely on actuators, or on filtering some commands sent by the control channel.

Safety properties: The safety monitor verifies a main property: the safety, i.e., the non-occurrence of hazards. The relevant safety properties to monitor can be elicited in different ways. In [Machin et al., 2018], as well as in [Sorin et al., 2016], the properties are identified during a risk analysis. From the list of hazards, the authors extract the list of safety invariants to be used for monitoring.

Some tools exist to automatically infer invariants from execution traces, and this concept has been used in [Jiang et al., 2017]. In this example, the authors show that they can automatically determine invariants such as “the drone can only land if the platform is horizontal”. However, this technique is only applicable in limited scenarios, as the system has to be extensively tested to generate the execution traces.

In other approaches, the properties are defined *ad-hoc*, as for instance in [Tomatis et al., 2003].

Safety rules synthesis: The authors of [Machin et al., 2018] automatically generate the rules. They model the known hazards to avoid along with the available observation and action means. A synthesis algorithm is then used to find the best association of actions to observed inputs to avoid reaching a hazardous state. This approach is further detailed in Section 1.3. The same authors explore a different approach in [Machin et al., 2015] using game theory for synthesizing safety requirements. This approach has however not been pursued for performances reasons. In [Bloem et al., 2015], the authors also use game theory to build a monitor finding the winning strategy between property violation and the deviation from the initial system’s output. The resulting mon-

itor will enforce some properties while deviating as little as possible from correct system behavior.

Other aspects of the monitors and their rules have to be considered:

Rules consistency. In an autonomous system, the safety rules can be numerous, and it is important to verify that no conflict between them exists, as pointed out in [Alexander et al., 2009]. Indeed, an autonomous car could have two rules stating that the car needs to brake whenever it is too close to the car in front of it, and to speed up whenever the car behind it is too close: these two rules are conflicting in some situations. A conflict between rules could introduce a new hazard or provide new ways to reach a hazardous state that was supposed to be avoided.

Balancing safety and functionality. Another concern that has rarely been explored for the definition of safety rules is the balance between safety and functionality (or availability). Complex systems typically have multiple and diverse tasks. The monitor must ensure safety while letting the system perform its tasks. An over-conservative monitor that would prohibit any movement may be safe but is obviously not very interesting in the context of autonomous systems (or any context). [Wagner et al., 2008] explore this issue for the Large Hadron Collider. However, the authors are trying to reduce the number of false alarms, and do not explicitly consider the reduction of availability due to the safety system's accurate intervention. Indeed, and it is the case for most electronic systems, the safety system's action is limited to shutting down the whole system. In the case of autonomous systems, such a drastic intervention is not desirable, as it can operate in a remote environment without a human operator nearby. Therefore, the monitor needs to ensure safety while limiting as little as possible the action means of the robot. In [Bloem et al., 2015], the authors model a *correctness* property — the system does not violate any rule — along with a *minimum interference* property — the monitor only changes the output of the control channel if necessary. The monitor is allowed to correct the control outputs for k consecutive steps. If another deviation is perceived within the k steps, the monitor does not try to respect the minimum interference property any more, but falls back to a *fail-safe* mode where only the correctness is ensured: the functionalities are degraded only in last resort. In [Kane and Koopman, 2013], the authors propose to allow transient violations of safety rules in order to avoid shutdowns as much as possible. They detail the use of a *soft-stop* as a first attempt to reach a safe state after a property violation. The soft-stop is implemented within the controller and will thus not be available is the failure comes from the controller itself. The soft-stop is followed by a reliable *hard-stop* if the system hasn't yet reached the expected safe state.

The authors of [Machin et al., 2018] model a *permissiveness* property, which verifies that the system can still reach a large set of states, guaranteeing that it can perform its tasks. This property however does not allow a lot of flexibility in its current version, and we will extensively discuss ways to adapt it the following chapters.

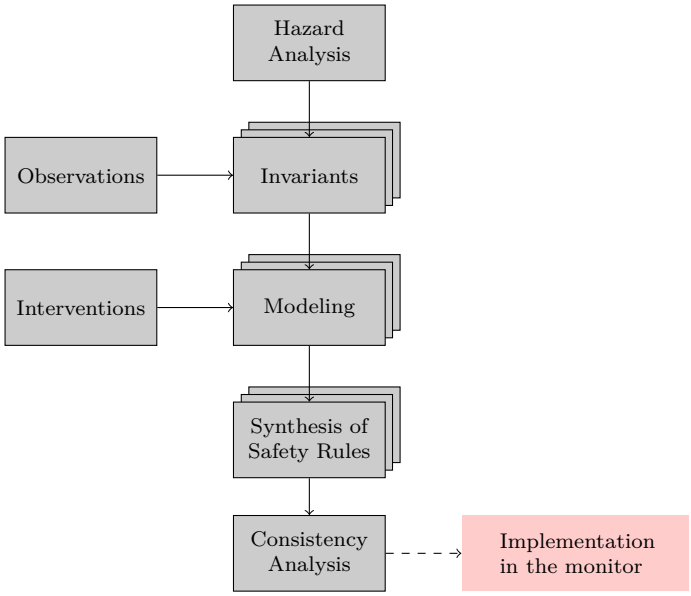

Figure 1.4: SMOF process

1.3 SMOF: Concepts and Tooling

From the diverse solutions presented before, the safety monitoring approach is a relevant candidate to keep an autonomous system in a safe state. We choose to adopt a reactive view: the monitor aims at preventing the reachability of catastrophic states, that are expressed by combinations of variable values. The monitor is assigned a high level of criticality and the logics it implements is kept simple. SMOF (Safety MOnitoring Framework) is a framework developed at LAAS, which we use and extend in our work. It provides a whole process to automatically synthesize safety strategies (using model checking) from invariants extracted from a hazard analysis. In this section we will explain the basic concepts of SMOF. Some examples will be shown in the next chapter where we will also discuss its limitations. For further information, the reader can refer to [Machin et al., 2018] and [SMOF, 2018].

1.3.1 SMOF Process Overview

The SMOF process overview is represented in Figure 1.4. The first step is to identify the potential hazards thanks to the HAZOP-UML analysis. From these hazards a list of invariants to be handled by the monitor is extracted. The invariants are then modeled: the observable variables are specified, as well as potential dependencies between them, the catastrophic state(s) and the available intervention(s). A template is available to ease the modeling, along with auto-completion modules. The synthesis algorithm provided by SMOF is launched in order to search for a suitable strategy. These same steps are followed for every invariant. Once a strategy is synthesized for each invariant, a

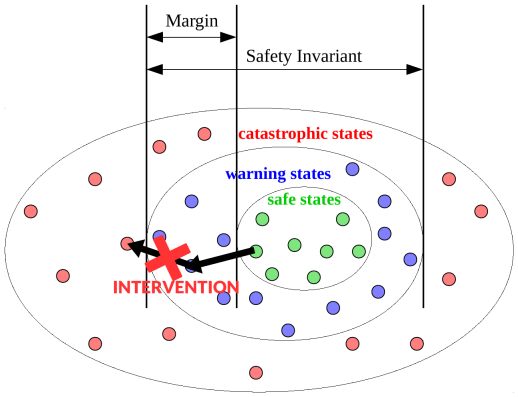

Figure 1.5: System state space from the perspective of the monitor

consistency analysis has to be performed, verifying that no conflict exists. The resulting set of rules can then be implemented in the monitor.

1.3.2 SMOF Concepts and Baseline

As a first step of the process (see Figure 1.4), one identifies a list of hazards that may occur during the system's operation, using the model-based hazard analysis HAZOP-UML [Guiochet, 2016]. From the list of hazards, one extracts those that can be treated by the monitor, i.e., the monitor has ways to intervene to prevent the hazard from occurring. These hazards are reformulated as *safety invariants* such that each hazard is represented by the violation of an invariant. A safety invariant is a logical formula over a set of observable variables. Formulating invariants from the list of hazards can highlight the need to provide the monitor with additional means of actions (mostly actuators), that we call *interventions*, or additional ways to evaluate the environment and/or internal state (mostly sensors), that we call *observations*.

A combination of observation values defines a system state, as perceived by the monitor. If one of the safety invariants is violated, the system enters a *catastrophic state* that is assumed to be irreversible. Each safety invariant partitions the state space into catastrophic and non-catastrophic states as represented in Figure 1.5 (non-catastrophic states being the warning (blue) and safe (green) states). The non-catastrophic states can in turn be partitioned into *safe* and *warning states*, in such a way that any path from a safe state to a catastrophic one traverses a warning state. The warning states correspond to safety margins on the values of observations.

The monitor has means to prevent the evolution of the system towards the catastrophic states: these means are a set of safety interventions made available to it. Most interventions are based on actuators, but they could also be software processes, e.g., processes filtering commands sent to the actuators. An intervention is modeled by its effect (constraint that cut some transitions) and preconditions (constraints on the state in which it can be applied). Interventions are applied in warning states in order to cut

all the existing transitions to the catastrophic states, as shown in Figure 1.5 by the red cross. Two types of interventions are identified in SMOF: safety actions that might change some state variables to put the system back into a safe state, and interlocks which forbid state variable changes (the system stays in a warning state).

The association of interventions to warning states constitutes a *safety rule*, and the set of all the safety rules constitutes a *safety strategy*. For example, let us assume that the invariant involves a predicate $v < V_{max}$ (the velocity should always be lower than V_{max}). In order to prevent evolution towards V_{max}, the strategy will typically associate a braking intervention to warning states corresponding to a velocity higher than the threshold $V_{max} - margin$. The determination of the size of the margin involves a worst-case analysis, accounting for the dynamics of the physical system, as well as for the detection and reaction time of the monitor after the threshold crossing.

1.3.3 Safety and Permissiveness Properties

The safety strategy must fulfill two types of properties: *safety* and *permissiveness* properties. Both properties are expressed using CTL (Computation Tree Logic) which is well suited for reachability properties. *Safety* is defined as the non-reachability of the catastrophic states. *Permissiveness* properties are intended to ensure that the strategy still permits functionality of the system, or, in other words, maintains its availability. This is necessary to avoid safe strategies that would constrain the system's behavior to the point where it becomes useless (e.g., always engaging brakes to forbid any movement). SMOF adopts the view that the monitored system will be able to achieve its tasks if it can freely reach a wide range of states (e.g., it can reach states with a nonzero velocity). Accordingly, permissiveness is generically formulated in terms of state reachability requirements: every non-catastrophic state must remain reachable from every other non-catastrophic state. This is called *universal permissiveness*. The safety strategy may cut some of the paths between pairs of states, but not all of the paths. In CTL, this is expressed as: $AG(EF(nc\_state))$, for each non-catastrophic state. Indeed, EF specifies that the state of interest is reachable from the initial state, and AG extends this to the reachability from every state.

The user can also use the *simple permissiveness* which merely requires the reachability from the initial state: $EF(nc\_state)$. It is much weaker than the universal permissiveness as it allows some of the monitor's interventions to be irreversible: after reaching a warning state in which the monitor intervenes, the system may be confined to a subset of states for the rest of the execution. For example, an emergency stop can permanently affect the ability of the system to reach states with a nonzero velocity.

1.3.4 SMOF Tooling

The SMOF tool support [SMOF, 2018] includes the synthesis algorithm and a modeling template to ease the formalization of the different elements of the model: the behavior model with a partition into safe, warning and catastrophic states; the available interventions modeled by their effect on observable state variables; the safety and permissiveness

properties. The template offers predefined modules, as well as auto-completion facilities. For example, the tool automatically identifies the set of warning states (having a transition to a catastrophic state). Also, the permissiveness properties are automatically generated based on the identification of non-catastrophic states. Finally, SMOF provides a synthesis tool based on the model-checker NuSMV [NuSMV, 2018]. For this reason the NuSMV language is used for the formalization. The SMOF synthesis tool relies on a branch-and-bound algorithm that associates interventions to warning states and checks some criteria to evaluate if the branch should be cut or explored. It returns a set of both safe and permissive strategies for the given invariant to enforce. The formalization and strategy synthesis is done for each invariant separately. Then a last step is to merge the models and to check for the consistency of the strategies selected for the different invariants. The SMOF method and tool have been applied to real examples of robots: an industrial co-worker in a manufacturing setting [Machin et al., 2018], and more recently a maintenance robot in airfield [Masson et al., 2017], which will be described in Chapter 2. Examples and tutorials can be found online [SMOF, 2018].

1.4 Conclusion

Autonomous systems are evolving towards more diversity of missions and tasks, and are expected to share their workspace with humans. Consequently, they are highly critical systems and require means to ensure their dependability. Because of their complexity and the unstructured environment they evolve in, some faults remain and need to be tolerated, and we saw that monitoring is an appropriate way to do so. We are particularly interested in independent safety monitors, as they constitute the ultimate barrier before the occurrence of a catastrophic failure.

An important issue when considering monitors for autonomous systems is the specification of the safety rules they implement. Indeed, they need to be defined using a dedicated method, that allows the association of the most appropriate reaction to a detected violation in order to cover an identified hazard. Due to the potentially high number of hazards to avoid, numerous rules can be synthesized and the consistency between them needs to be verified, as well as their implementation on the actual system.

Also, the introduction of a safety monitor may have an impact on the system's functionalities that needs to be identified and moderated. This is related to the choice of the recovery actions made available to the monitor: we saw that shutting down the system is not satisfying in the context of autonomy. This issue is not sufficiently addressed by the research community.

The framework SMOF, previously developed at LAAS, partially solves this problem by proposing a synthesis algorithm to automatically generate safety rules. However, it still suffers from limitations. The identification of such limitations is done in the next chapter, based on several cases studied.

Take Aways:

- Autonomous systems are highly critical systems because of their inherent complexity and the unstructured environment they evolve in;
- Fault-tolerance has to be considered, as the complete removal of faults is not feasible;
- Independent safety monitors are an appropriate way to ensure the safety of autonomous systems;
- The specification of safety rules for monitors is challenging: they need to ensure safety while reducing as little as possible the system's ability to perform its tasks;
- SMOF has been introduced to solve the problem of synthesizing safety rules for independent active safety monitors. In this work we focus on cases where SMOF fails to return a satisfying solution.

Chapter 2

Feedback from the Application of SMOF on Case Studies

Contents

SMOF (Safety MOnitoring Framework) has been developed to solve the issue of defining safety rules for active safety monitors. It is a process for automatically synthesizing safety rules from hazards identified during a hazard analysis.

The first objective of this chapter is to illustrate the SMOF method on a complete case study from the company Sterela. We will see that for complex examples, several iterations may be necessary to be able to synthesize a solution when none was found with the initial requirements. The second objective is to identify the cases where synthesis fails to return a satisfying set of rules and to analyze which features of SMOF are required to find solutions.

In Section 2.1, we will present and analyze the results of a case study provided by the company Sterela [Sterela, 2018], as part of the European project CPSE-Labs [CPSE-Labs, 2018]. To complete our analysis we will revisit some examples from a former experiment in Section 2.2, performed as part of the European project SAPHARI [SAPHARI, 2018], on a robot from the German company Kuka [KUKA, 2018]. The lessons learned from these two experiments are presented in Section 2.3, where we will present the identified limitations of SMOF, and explain the solutions that were manually

implemented for them. We finally draw conclusions on these case studies and highlight the needs for tools to assist the user in the task of finding a solution when the synthesis fails to return a satisfying one (Section 2.4).

2.1 Sterela Case Study

In this section we apply SMOF to a maintenance robot from the company Sterela. We first identify hazards using a hazard analysis called HAZOP-UML. From these hazards, we identify a list of invariants to be handled by the monitor. These invariants are then modeled, along with the available interventions and desired safety and permissiveness properties. SMOF synthesis is finally run to find satisfying safety strategies. When no strategies can be found, the user needs to change the defined properties, observations or interventions. SMOF is applied at an early stage where some design decisions are still open and can be tuned to accommodate safety concerns. Indeed, this work could be used by Sterela to revise the design of their robot. Several iterations may be necessary before being able to synthesize a satisfying solution.

2.1.1 System Overview

The studied use case concerns maintenance and control of the lights along airport runways. The International Civil Aviation Organization (ICAO) recommends monthly measurement of the light intensity of airfield lighting installations, using a certified device. If the light intensity does not comply with the ICAO requirements, no air traffic can be allowed on the airfield.

Currently, human operators perform the measurements but this is a burdensome and displeasing task. It has to be done late at night, typically from 1 a.m. to 4 a.m. The repeated exposure to intense lights in the surrounding dark may cause eye strain. The task is thus a perfect fit for a robotic operation.

The French company Sterela is developing a prototype system to serve this purpose: its mission is to move around the workspace to perform some maintenance tasks, especially the measurement of the lights' intensity. The robot consists of a mobile platform and a commutable payload (Figure 2.1). The platform, called 4MOB, is a four-wheel drive vehicle. The payload for the light measurement task is a certified photometric sensor on a support deported on the side of the platform. The deported sensor moves above the lights (15-20 cm) with a maximum speed of 1.4 m/s. A human operator is supervising the mission with a digital tablet from the extremity of the runway. As the robot operates at night and at long distances, the operator has no direct visual contact with it.

Our goal is to specify a safety monitor for this robot to tolerate faults that may occur during its operation.

Figure 2.1: Sterela robot measuring runway lamps

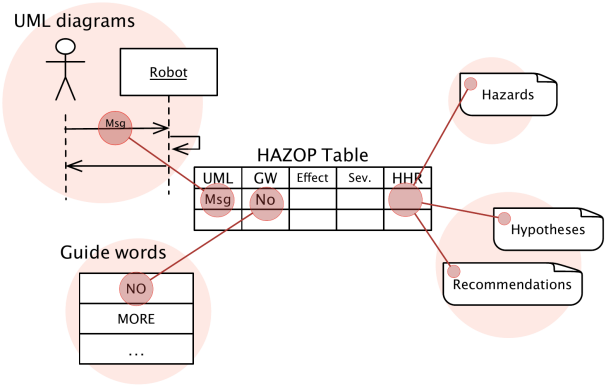

Figure 2.2: HAZOP-UML overview

2.1.2 Hazop-UML Analysis

Several important hazards can exist for such an autonomous robot on real airports. Other operators and service vehicles could be present on the runway on which the robot evolves, and the robot must not constitute a hazard to them. Moreover, even if the robot could be allowed on runways, there would be airfield areas strictly forbidden to it: it must never traverse them. Also, the air traffic controllers should keep the ability to re-open a runway as quickly as possible in case of a landing emergency.

To properly identify these hazards, we apply the HAZOP-UML technique. It is a model-based hazard analysis technique developed at LAAS [Guiochet, 2016]. It combines the well-known modeling language UML (Unified Modeling Language) and a hazard analysis technique called HAZOP (HAZard OPerability). As presented in Figure 2.2, based on the UML models and many predefined guide words, a systematic deviation analysis is performed.

Let us consider for example the nominal scenario of a mission: the robot has to check the lights of runway number 1 (see UML Sequence Diagram in Figure 2.3). The operator loads the map in the robot and defines its mission (here checking lights of the runway 1)

before starting it. The robot then drives itself to the beginning of the runway, performs the measurements on both sides, and when it is done drives itself back to the base (the storage area). The operator can then load the results of the measurement.

An extract of the Hazop table resulting from the analysis is shown in Table 2.1. For instance, we analyzed the [1: loadMap()] message, associated with the guide word *other than*: the operator loads the wrong map. This is highly critical as the robot could calculate an itinerary according to an erroneous map, therefore not being able to conduct the mission, or worse, driving itself to prohibited areas.

Being a systematic approach, HAZOP-UML typically induces the analysis of many potential deviations. Here, a total of 1623 deviations were analyzed (see Table 2.2 for statistics). A first way to control complexity is by setting the level of detail of the UML models. This has been done through the number of UML elements (47 messages and 24 conditions). A second way is to provide tool support that alleviates the effort. For this, Sterela has extended their requirement editing tool to partially generate UML diagrams and HAZOP tables. This tool is connected to their application lifecycle management software based on the open source Tuleap platform [Tuleap, 2018]. In parallel, a dedicated open-source tool called *Youcan* has been developed at LAAS, based on Eclipse-Papyrus [HAZOP-UML, 2018].

As a first result of the study, a list of recommendations for the use or the design of the robot is produced. For example, an evacuation procedure was initially considered in case of an emergency landing. The analysis showed that such a critical aspect cannot be trustworthy, as it would mean certifying the complete functional chain for the evacuation: the navigation system, the battery charge, the sensors and actuators, etc. Indeed, the battery charge for example can not be observed precisely, therefore the robot could run out of battery level while evacuating the runway. Instead we propose to have a high integrity on the robot's position, so that the operator can go get it and proceed to a manual evacuation of the runway.

A second result is the identification of the hazards to be treated by the safety monitor. Ten highly critical hazards are extracted from the HAZOP analysis and presented in Table 2.3. Some of them rely on operating rules: for instance, the control tower has to validate the runway that will be checked to make sure no other activity will take place on it (other checks, landings, etc.). Only a subset of hazards is in the scope of the monitor.

For some hazards in the scope of the monitor, it is possible to directly extract some safety invariants. For instance, the hazard **H9** (movement in an unauthorized zone) can be directly formalized from available observations (distance to the unauthorized zone). But for other hazards, it is not possible due to a lack of observations or interventions. In such cases, an in-depth analysis of the hazard is performed in order to identify the premises leading to the hazard and translate them into safety invariants. It may not be possible to translate all of the premises of a hazard into a safety invariant. No method has yet been defined for the analysis of the hazard and its premises, and their reformulation into safety invariants. It has to be done manually and relies on the expertise of the user.

For the studied robot, the monitor may address 4 hazards (out of 10): **H6**, **H8**, **H9** and **H10**. The derived invariants are the following:

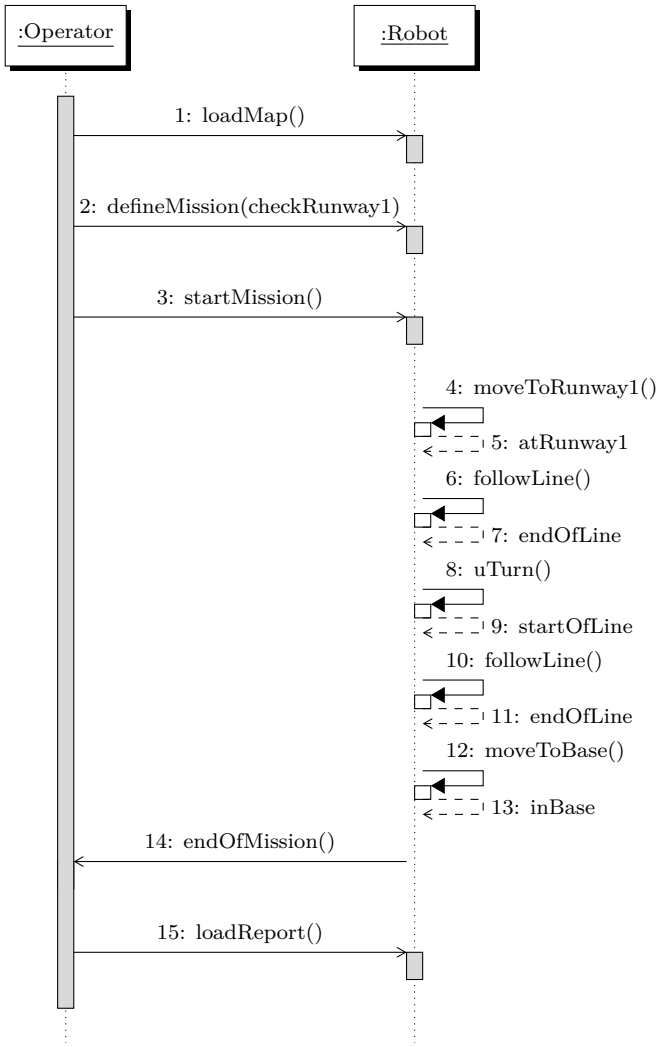

Figure 2.3: UML Sequence Diagram for a light measurement mission

Entity	**Attribute type**	**Guide word**	**Deviation**	**Effect**	**Severity**	**Safety recommendation**
[1: loadMap()]	Message	Other than	The wrong map is loaded	The robot calculates its trajectory using a erroneous map: it may drive through a prohibited zone	High	Marking of the prohibited zone to enable their detection even with erroneous map + monitoring the distance to the prohibited zone
[4: moveToRunway1()]	Message	Other than	The robot moves to another runway	The robot may start checking the lights of a runway that is open for landing: risk of collision with planes	High	The control tower has to validate the mission + the operator needs to control that the robot is checking the expected runway
[10: followLine()]	Message	No	The robot doesn't follow the line	The robot doesn't drive parallel to the line of lights: it may collide with one	High	Embed a collision avoidance mechanism

Table 2.1: Hazop table extract for light measurements missions

	4MOB
Use cases	5
Conditions (pre,post,inv)	24
Sequence diagrams	5
Messages	47
Deviations	1623
Interpreted deviations	226
Interpreted deviations with severity > 0	97
Number of hazards	10

Table 2.2: Statistics for the application of HAZOP-UML to the 4MOB robot

Number	Hazard	Comments
H1.	Runway open to planes	Operating rule
H2.	Battery charge not sufficient	Operating rule
H3.	Lights not correctly verified	Operating rule
H4.	Operator unavailable	Operating rule
H5.	Debris deposit on the runway	Mechanical structure
H6.	Dangerous velocity	Monitor
H7.	Fall of the robot during loading/unloading	Operating rule
H8.	Impossibility to free the runway	Monitor
H9.	Movement in unauthorized zone	Monitor
H10.	Collision of the robot with an obstacle	Monitor

Table 2.3: Hazards extracted from the HAZOP analysis

SI_1 Dangerous velocity—the robot must not go too fast in terms of linear or angular velocity;

SI_2 Impossibility to free the runway—the location of the robot must be known at any time;

SI_3 Movement in an unauthorized zone—the robot must not enter a prohibited zone;

SI_4 Collision with an obstacle (including lights)—the robot must not collide with an obstacle.

HAZOP-UML is particularly well adapted to the context of safety monitoring, since it focuses on operational hazards. The interactions with the environment related to the mission are analyzed. Other hazards such as electric shocks, sharp edges, etc., are treated by other means.

2.1.3 Modeling and Synthesis of Strategies for SI_1, SI_2 and SI_3

In this section we detail the modeling and the synthesized strategies for the invariants SI_1 (the robot must not go too fast), SI_2 (the location of the robot must be known at any time) and SI_3 (the robot must not enter a prohibited zone). These invariants are simple and the strategies fairly straightforward, but they already raise interesting points of interest. They also allow us to detail the SMOF method.

Details of SMOF Method and Application to SI_1: Dangerous Velocity

The first safety invariant (SI) to be covered by the monitor is *SI_1 the robot must not exceed a maximum velocity v_{max}*. In this section, we use this simple invariant as a running example for detailing the different steps of the SMOF process.

A list of SIs has been extracted from the results of the hazard analysis and the SIs are expressed in natural language (here, "the robot must not exceed a maximum velocity v_{max}"). Each SI is then expressed formally with predicates on variables that are observable by the monitor. We focus on predicates involving variables compared to fixed thresholds. For our running example, we consider only one observable variable: the robot's velocity v, which is evaluated compared to the safety value v_{max}. The SI is formalized as $SI = v < v_{max}$. The negation of this invariant defines the catastrophic states (the ones where $v \geq v_{max}$). Note that this step may induce an early feedback on the system design, by revealing the lack of key observation mechanisms.

The SMOF modeling template is then used to build state-based models. In order to keep models simple enough to be validated, each SI is modeled separately.

A SMOF model formalizes the part of the system related to one SI, seen from the monitor's point of view. It gathers all information necessary to produce strategies that ensure the SI:

- The *behavior*: the automaton of the system safety-related state variables in absence of the monitor, containing all paths to the catastrophic states;

- The *interventions*: the abilities of the monitor to constrain the system behavior;
- The *safety* and *permissiveness* properties: desired properties of the monitor action.

Behavior: A safety invariant is actually a condition composed of state variables and constants. The comparison defines a first partition of the variables into classes of values, e.g., the values that are below or above a safety threshold. The variable v is compared to the safety threshold v_{max}, i.e., $v < v_{max}$ or $v \geq v_{max}$.

The partition is then refined in order to consider margins. In our example, a margin is taken on the velocity variable, i.e., the velocity is either lower than the margin ($v < v_{max} - margin$), within the margin ($v_{max} - margin \leq v < v_{max}$) or above the maximum value ($v \geq v_{max}$). SMOF encompasses a modeling template for the definition of the behavior model, that eases the modeling in NuSMV, which we use for the strategy synthesis. The resulting classes of values are encoded by integers, i.e., the velocity variable is encoded by the discrete variable $\texttt{v} \in \{[0, v_{max} - margin[, [v_{max} - margin, v_{max}[, [v_{max}, \infty[\}=\{\texttt{0,1,2}\}$.

The user must also enter a definition of the catastrophic state expressed with the discrete variables encoding the partition (`cata: v=2`).

The behavior states result from the Cartesian product of the ranges of all discrete variables. The user does not need to list the states: NuSMV automatically builds the state space from the declaration of variables.

By default, all the combinations of variable values (i.e., states) are possible and all transitions between pairs of states are implicitly declared. The user can add constraints to delete states and transitions that would be physically impossible. The most common constraint is the "continuity" of a variable, e.g., the velocity cannot "jump" from `0` (i.e., the velocity is lower than the margin) to `2` (i.e., the velocity is above the maximum allowed value). Such a continuity constraint is encoded in SMOF using the dedicated module `Continuity` of the modeling template, that encapsulates the constraint `next(x)= x | x+1 | x-1`, i.e., the next value of `x` can stay the same, or increase or decrease by one. For the velocity variable of our example, this is encoded as follows:

```
--Continuity(value_min, value_max, value_init)
VAR v :  Continuity(0, 2, 0);
```

The user can also declare dependency constraints. For instance, she can specify that a position variable cannot evolve if the velocity variable is zero.

The catastrophic states are considered irreversible. The user does not need to explicitly specify this: the constraint is automatically included in the SMOF modeling template, where all the catastrophic states are sink states.

The resulting behavior model for the running example is presented in Figure 2.4. It only has three states: one safe state (`v=0`), one warning state (`v=1`) (we call warning states those that lead to a catastrophic state in one step) and one catastrophic state (`v=2`).

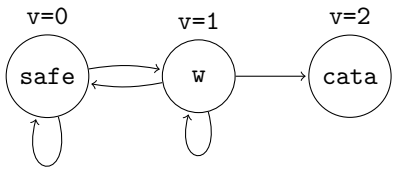

Figure 2.4: Behavior for the invariant SI_1.

Interventions: The interventions are the means for the monitor to intervene on the system to avoid reaching a catastrophic state.

An intervention is modeled by its effect and preconditions. The effect is a constraint that cuts some transitions from the state in which the intervention is applied, reducing the set of possible next states.

The effect is guaranteed only if the preconditions are verified. We distinguish two types of preconditions. The *static preconditions* are constraints on the state in which the intervention is applied. For instance, an intervention preventing an arm from extending can only be effective if the arm is not already extended, thus, the precondition would be "the arm is folded".

The effectiveness of an intervention may also depend on the system history. We only take into account the state preceding the application of the intervention. We consider *sequential preconditions* that must hold on this state. They are constraints on the transitions that trigger the intervention, when moving from a state in which the intervention is not applied to a state in which it is. For our running example, only one intervention influencing the variable v is available: triggering the brakes. Its effect is to prevent the velocity from increasing. It can be encoded as `next(v)!=v+1`, i.e., `v` does not increase. Due to the inertia of the platform, this intervention may not be efficient immediately: this is modeled with a sequential precondition. For the braking intervention to be effective, it is necessary that the threshold $v_{max} - margin$ has just been crossed (`v=0` in the previous state). Indeed, if the threshold has been crossed for more than one step, the real value of the velocity could be $v_{max} - \epsilon$, and it would be too late to brake, the value v_{max} would be exceeded and the invariant violated.

The SMOF template provides a predefined `Intervention` module that can be used as follows to model the braking intervention (a `TRUE` value indicates that no precondition is declared; the precondition is always satisfied):

```
-- Interv(stat.  precond., seq.  precond., effect)
VAR brake:  Interv(TRUE, v=0, next(v)!=v+1);
```

Safety and permissiveness properties : They are modeled in computation tree logic (CTL), a branching time logic fully supported by NuSMV. A CTL operator is composed of one branching operator (A, all the branches, or E, there exists a branch) and one time operator (X for the next state, G for the entire path or F for eventually). It is applied on states, or more generally on statements about the system state.

The safety property is predefined as the unreachability of the catastrophic states declared by the user, i.e., in CTL, $AG(\neg cata)$.

The permissiveness property aims to ensure that the system can still perform its tasks. In SMOF, it is considered that a versatile autonomous system has to operate in many different states. Hence, permissiveness is expressed in terms of state reachability properties: the monitored system should keep its ability to reach non-catastrophic states. More precisely, permissiveness is modeled by two reachability properties applied to any non-catastrophic state s_{nc}:

- Simple reachability ($EF(s_{nc})$): the state s_{nc} is reachable from the initial state;
- Universal reachability ($AG(EF(s_{nc}))$): the state s_{nc} is reachable from any reachable state. A state that is universally reachable is also simply reachable.

The SMOF template has a module that automatically generates the simple and universal permissiveness properties for every non-catastrophic state. The universal and simple permissiveness are required by default, but the user can change it to require only the simple permissiveness.

Summary of the modeling: The SMOF template includes predefined modules, parts to be edited by the user, and generated parts. For the running example, the user has only to model three lines:

```
--Declaration of the variables
--Continuity(value_min, value_max, value_init)
VAR v :  Continuity(0, 2, 0);
--Declaration of catastrophic states
DEFINE cata := v=2;
--Declaration of the interventions
--Interv(stat.  precond., seq.  precond., effect)
VAR brake:  Interv(TRUE, v=0, next(v)!=v+1);
```

The permissiveness properties are automatically generated. The tool also generates the list of warning states, i.e., of states having a transition to a catastrophic state. This is done in preparation for the synthesis of strategies: the warning states are candidate states for the application of interventions.

In our running example, there is only one warning state, whose definition is generated automatically ("flag" denotes the definition of the state):

```
DEFINE flag_st_1:=v=1;
```

Synthesis of strategies: The synthesis algorithm takes as inputs the behavior model for an invariant, the available interventions and the properties to ensure (safety and permissiveness). It outputs a set of alternative strategies, each of them satisfying the properties.

Conceptually, the strategies assign a combination of interventions to each warning state, cutting the transitions to the catastrophic states. This is encoded by the definition of flags in the model, i.e., triggering conditions for the interventions.

Instead of enumerating all the possible strategies, which would be very inefficient, the SMOF synthesis algorithm builds a tree of strategies, making it possible to prune branches during the search (branch-and-bound algorithm). It uses several pruning criteria to evaluate if a branch is worth exploring or needs to be cut.

Three variants of the algorithm exist, using more or less pruning criteria. The first one gives an exhaustive list of the satisfying strategies. This variant comes with an optional post-processing algorithm that sorts the strategies to only return the *minimal* strategies. A strategy is minimal if removing one intervention means violating the invariant, i.e., all the interventions are absolutely necessary to ensure the safety property. The second variant gives a superset of the set of minimal strategies: all the minimal strategies, plus some non-minimal ones.

The two first variants can take a long time to execute, as they explore all the minimal strategies plus others. However, if they do not return any solution, then it is sure that no solution exists for the model. A third variant exists, that gives a subset of minimal strategies. It is not conclusive regarding the absence of solutions, but may return minimal strategies with a shorter tree traversal than the two other variants. This variant is very efficient for most cases, but it sometimes is too drastic, especially in the case of the existence of sequential preconditions. In these cases, all satisfying strategies may be cut during the exploration. For a detailed presentation of the synthesis algorithm and its different variants, the user can refer to [Machin et al., 2018].

For our running example, a single strategy exists:

```
DEFINE flag_st_1:=v=1;
-- Strategy #1
DEFINE flag_brake := flag_st_1;
```

As anticipated, the strategy found by SMOF triggers the brakes whenever the velocity reaches a value within the margin (`v=1`). This example is very simple, but useful to illustrate the approach. The invariants detailed thereafter present more interesting features.

When the model does not admit a satisfying strategy, the user has several options. They can reduce the permissiveness, requiring only the simple permissiveness instead of the universal permissiveness. They also can add new interventions, which will have to be implemented in the real system.

SI_2: Impossibility to Locate the Robot

The invariant SI_2 derives from the danger **H8**: it is impossible to free the runway. A solution would be to trigger an evacuation procedure: the robot drives itself to a fallback zone. As mentioned in Section 2.1.2, this would require a high integrity level on the navigation, the battery charge, all the control chain including sensors and actuators,

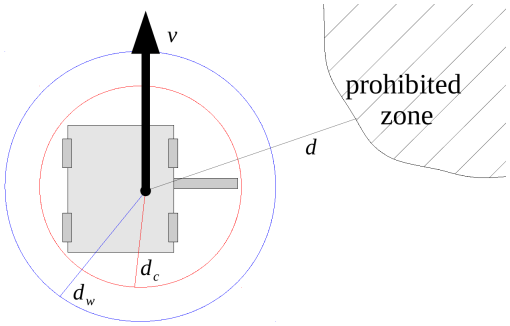

Figure 2.5: SI_3 the robot must not enter a prohibited zone

which is not realistic. Therefore, we propose another way to avoid this danger: in case of an emergency landing, the operator has time to go get the robot and take it out of the runway. In order to do this, the operator needs to know the robot's location (the robot is operating at night, and the operator has no direct visual contact with it). The robot is communicating its location regularly to the operator. The location as well as the communication are considered to have a high integrity level, i.e., the transmitted location data are correct. This means that only the broken communication could prevent the operator from knowing the location of the robot. We can thus reduce the danger to the following: the robot moves without the communication being operational.

The corresponding invariant is *SI_2' the robot must not move if the communication is not operational*. We model this through two observations: the time since the loss of the communication, and the velocity of the robot (the robot is stopped or moving). The available intervention is a full stop intervention that stops the robot completely. SMOF synthesis returns one strategy, which is to trigger the full stop as soon as the communication is lost.

SI_3: Movement in a Prohibited Zone

The considered invariant here is: *SI_3 the robot must not enter a prohibited zone*. We chose to use the observation d, the distance to the prohibited zone (see Figure 2.5). The distance is calculated from the location of the robot and the position of the prohibited zones (that is both marked and stored in the robot's memory). This distance variable is partitioned according to the concept of margin: `d` $\in$ {`0, 1, 2`}, `0` representing the robot into the prohibited zone ($d < d_c$), `1` the robot close to the prohibited zone ($d_c \leq d < d_w$) and `2` the robot far from the prohibited zone ($d \geq d_w$). According to this partition, the catastrophic state can be expressed as `cata:= d=0`.

The only available intervention here is the *full stop* intervention, which stops the robot completely. To model this intervention, we use the velocity `v`, partitioned as follows: `v` $\in$ {`0, 1`}, i.e., the robot is either stopped (`v=0`) or moving (`v=1`). It is necessary to model the full stop with a velocity variable: it is not possible to model it with the

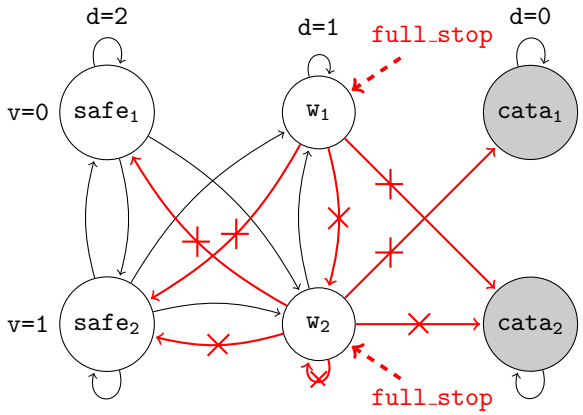

Figure 2.6: Expected strategy for SI_3

distance variable only. Even if the distance to the zone stays the same, it does not mean that the robot is stopped: the robot could move parallel to it.

The velocity and distance variables are dependent: the evolution of one depends on the evolution of the other. Indeed, the distance cannot change if the robot is not moving. We specify this requirement with a dependency constraint. Is is encoded in SMOF as `TRANS v=0 & next(v)=0 → next(d)=d`.

The full stop intervention is only effective under the precondition that the distance threshold to the prohibited zone has just been crossed (`d=2` in the previous state), and affects the velocity variable `v` (`next(v)=0`). It is encoded as follows (remember we write an intervention with the `Interv(stat_precond, seq_precond, effect)` template):

```
|| VAR full_stop:  Interv(TRUE, d=2, next(v)=0);
```

Considering that only the full stop intervention is available, the expected solution —displayed in Figure 2.6— would be to trigger it whenever the robot is getting too close to the prohibited zone. However, when we run the SMOF synthesis on this model the tool returns no solution with the default universal permissiveness property. Indeed, the universal permissiveness, expressed in CTL as $AG(EF(s_{nc}))$ for every non-catastrophic state, requires the reachability of every non-catastrophic state from every other non-catastrophic state. In our case, the full stop freezes the distance to the prohibited area: if it is triggered when `d=1` the robot will be stopped and blocked at its current location (in states w_1 and w_2), i.e., the states where `d=2` will no longer be reachable.

We know that a safe strategy exists, the one we just discussed (it can be found by SMOF when removing all the permissiveness constraints for the synthesis): this means that the universal permissiveness (the default one for SMOF synthesis) could not be satisfied. Then, to synthesize the expected strategy, SMOF permissiveness needs to be switched to simple permissiveness. Simple permissiveness is expressed as $EF(s_{nc})$: every non-catastrophic state has to be reachable from the initial state. This allows the monitor's intervention to be irreversible (the system is stuck in states w_1 or w_2). Every

non-catastrophic state can be reached from the initial state, but after an intervention is triggered, some states may no longer remain reachable, as is the case here, the robot being blocked close to the prohibited zone. Concretely, this implies that a human intervention is then needed to restart the robot (after moving it to an authorized area).

Switching to simple permissiveness allows in some cases to synthesize a strategy when none was found with the universal permissiveness. However, this does not give a good understanding of the restriction that is imposed by the resulting strategy. No precise indication is given concerning the reachability of the states. The user is not able to differentiate a strategy where all the states are reachable from every other state except for one (universal reachability of all the states but one), from a strategy where all the states would only be reachable from the initial state (simple reachability of all the states). In such a simple case as the one presented above, this is not a big issue as the graph can be represented and manually analyzed, but we foresee that it can be an issue with more complex and larger models.

☞ There is no solution with the universal permissiveness requirement. We needed to weaken the permissiveness into simple permissiveness. In the general case, it is hard to assess the concrete impact of this operation on the robot's functioning, i.e., on the reachability of the states.

Note that we use the symbol ☞ to denote the main points of interest identified when applying SMOF. They will be summarized and discussed in Section 2.3.

2.1.4 Modeling and Synthesis of Strategies for SI_4

The absence of collision is an important problem for the studied robot. To get a proper light measurement, the robot should be allowed to move very close to the lights, so that its deported sensor passes above them (see Figure 2.1). Still, it must not collide with them. Also, it must not be allowed to move close to other types of obstacles like humans or service vehicles.

Due to its complexity, the formalization of the problem was done in two steps. We first considered a simple case, with a single idealized obstacle modeled by a pair of (x, y) coordinates called the Cartesian model. It helped us to identify some assumptions on the obstacles to address, as well as some requirements on how to observe the neighborhood of the robot and which interventions to provide. The second model, called the "zones model" was more complex, accommodating multiple obstacles and their spatial extension. It allowed us to confirm the strategy synthesized from the simple case.

Cartesian model: We assume that the robot is not responsible for rear-end collisions (like in car accidents). It is then sufficient to monitor the neighborhood in front of, and at the sides of the robot. However, the robot must not be allowed to move backwards, or to make sharp turns (remember it is with four-wheel steering, it might turn 360 ° on the spot). Such movements would be dangerous for stationary obstacles close behind it. To avoid this, we add two invariants (not presented here) that restrict the direction and curvature radius of movement.

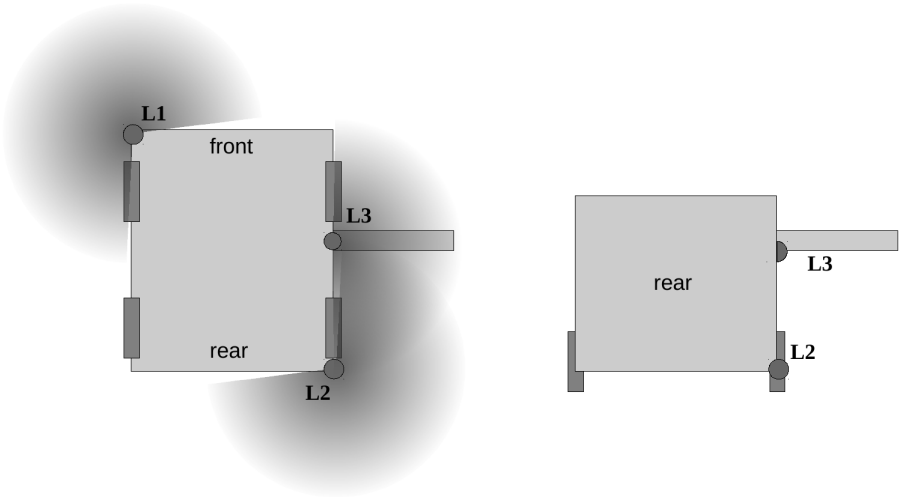

Figure 2.7: Disposition of the lasers on the robot

Initially, two laser sensors were implemented on the bottom of the robot platform. These sensors detect everything on a plane more than 180 ° around them. Thanks to them, the robot is able to detect the presence of an obstacle. They are positioned as shown in Figure 2.7 (laser sensors L1 and L2). However, these sensors are not sufficient to identify the size of the obstacles: the robot cannot differentiate between low obstacles (20 cm max), which can pass under the deported sensor, or high obstacles, including humans which are too tall to pass under it. In this case, the only way to make sure that no obstacle higher than the deported sensor collides with it is to prevent any obstacle from getting close to it. This would mean that no obstacle, including lights, would be allowed close to the sensor, hence it would not be possible to measure the lights.

The robot must be able to differentiate between low and high obstacles, at least for obstacles that are located on the same side as the deported sensor. The chosen solution was to equip the robot with an additional 2D Lidar unit on the deported extension (L3 in Figure 2.7). The size of the obstacle (high or low) can then be inferred by observing the combination of sensor values. The low obstacles, including lights, are assumed stationary: otherwise, there would be no safe strategy allowing the robot to move close to them.

☞ A sensor was added to make it possible for the robot to perform the light measurement.

The monitor can use the following observations:
x : abscissa of the obstacle in the robot's referential
y : ordinate of the obstacle in the robot's referential
v : robot's velocity
$type$: type of the obstacle (high or low)

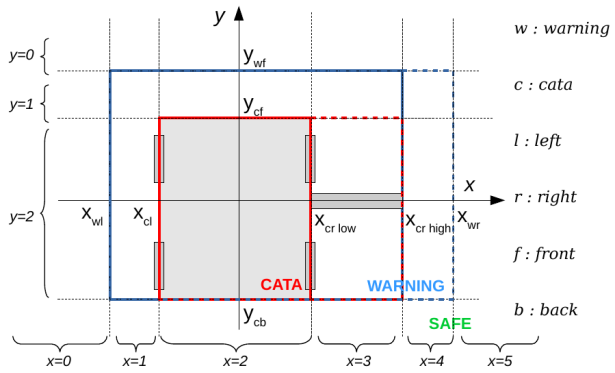

Figure 2.8: Visual representation of the classes of `(x,y)` coordinates

For x		**For** y	
$x < x_{wl}$	`x=0`	$y > y_{wf}$	`y=0`
$x_{wl} \leq x < x_{cl}$	`x=1`	$y_{cf} < y \leq y_{wf}$	`y=1`
$x_{cl} \leq x \leq x_{crlow}$	`x=2`	$y_{cf} \leq y \leq y_{cb}$	`y=2`
$x_{crlow} < x \leq x_{crhigh}$	`x=3`		
$x_{crhigh} < x \leq x_{wr}$	`x=4`		
$x > x_{wr}$	`x=5`		
For v		**For the** *type* **(when `x>2`)**	
The robot is standstill	`v=0`	The obstacle is low	`type=0`
The robot is moving	`v=1`	The obstacle is high	`type=1`

Table 2.4: Partitioning and abstraction of the variables for the Cartesian model

As for the previous invariants, their values are partitioned and discrete variables are introduced to represent them. Figure 2.8 gives a visual representation of how the space around the robot is partitioned, and Table 2.4 recaps the encoding for all variables.

The `(x,y)` coordinates are declared with continuity constraints. An additional constraint models the stationary assumption of low obstacles (having `type=0`). However, the type of obstacle is observed only on the right side of the robot (`x>2`). In other location areas, the type is forced to the default value `1` (the obstacle is potentially high and mobile).

The catastrophic states are collisions with the platform (for any type of obstacle) or with the deported sensor (for a high obstacle) at a non-zero velocity. Indeed, we consider that it is not dangerous to have a collision when the robot is at a standstill: the operator could get close to the robot and touch it. Using the encoding into abstract variables, `cata` is defined as `v=1 & (x=2 & y=2 | type=1 & x=3 & y=2)`.

One intervention is needed that is able to stop the robot completely: the full stop. It is redefined for these variables. It stops the platform under a threshold crossing

precondition, i.e., the obstacle was previously far away on the left (`x=0`), on the right (`x=5`) or in front (`y=0`) of the robot. The threshold distance has to be calculated considering the real speed and braking distance parameters. An operating rule states that no vehicle can be on the runway when the robot is in operation. We only have to consider pedestrians, with a speed of around 5 km/h. The distance threshold then is such that the robot can stop before reaching an obstacle that is potentially moving at a maximum of 5 km/h. Sterela has computed the concrete numerical value of the threshold. In the SMOF model, the variables and interventions are kept abstract. The declaration is written as follows:

```
-- Interv(stat_precond, seq_precond, effect)
VAR full_stop:  Interv(TRUE, x=0 | x=5 | y=0, next(v)=0);
```

If this intervention is the only one available, a safe strategy exists but it is not permissive (the synthesis returns no safe and permissive solution, but a safe solution is found when the permissiveness requirements are removed). The safe strategy that can be synthesized is as follows: the robot stops whenever any type of obstacle enters the warning zone. But to measure the lights the robot needs to move close to a low obstacle (i.e., `v=1 & x=3`). The set of states satisfying the predicate `v=1 & x=3` are warning states with regard to the collision invariant: the obstacles are close enough to collide with the platform in one step. However, these states need to be reached to satisfy the permissiveness constraints (the robot can measure lights). This case study is the first time we face a case where warning states are operational states. Warning states are states that are "close" to the catastrophic states (the catastrophic states can be reached in one step from the warning states). In most cases, being close to the catastrophic states is not desirable for the normal operation of the robot (for instance, approaching an excessive speed is not necessary for the robot's movement). The warning states can be exited as soon as they are reached (the interventions make the robot return to a safe state). However, here the goal is to stay in the warning states where a lamp is under the arm while blocking the evolution towards the catastrophic state (i.e., collision).

In order to ensure permissiveness, we manually determined that a second intervention is needed, which would be able to block the evolution from the warning states where a lamp is under the arm to the catastrophic states. A satisfying intervention is to restrict the moves of the robot in its direction to the right. The robot will not be able to get any closer to the lights (on its right side). It is effective for stationary obstacles only, that is, for low obstacles only under our modeling assumption.

```
-- Interv(stat_precond, seq_precond, effect)
VAR restrict_right_curve:  Interv(type=0, TRUE, next(x)!=x-1);
```

This intervention was manually defined and several iterations and discussions were necessary to achieve a satisfying result both respecting the specification (preventing the collision with lamps under the arm) and being implementable (Sterela engineers needed to be able to implement the intervention on the robot with a high level of integrity). This was complicated and some help on the definition of this intervention would have been useful.

With the two above interventions, the synthesis returns one safe and permissive solution: trigger the full stop when a high obstacle is close, irrespective of whether the robot is moving or not, and restrict the curve when a low obstacle is in the deported sensor zone and the robot is moving.

☞ No strategy is possible if the full stop is the only intervention available. An additional intervention had to be designed.

The strategy generated by SMOF is as follows:

```
--Definition of the warning states
DEFINE flag_st_0 := x=3 & y=1 & v=1 & type=0 ;
DEFINE flag_st_1 := x=3 & y=2 & v=1 & type=0 ;
DEFINE flag_st_2 := x=1 & y=1 & v=0 & type=1 ;
DEFINE flag_st_3 := x=2 & y=1 & v=0 & type=1 ;
DEFINE flag_st_4 := x=3 & y=1 & v=0 & type=1 ;
DEFINE flag_st_5 := x=4 & y=1 & v=0 & type=1 ;
DEFINE flag_st_6 := x=1 & y=2 & v=0 & type=1 ;
DEFINE flag_st_7 := x=2 & y=2 & v=0 & type=1 ;
DEFINE flag_st_8 := x=3 & y=2 & v=0 & type=1 ;
DEFINE flag_st_9 := x=4 & y=2 & v=0 & type=1 ;
DEFINE flag_st_10 := x=1 & y=1 & v=1 & type=1 ;
DEFINE flag_st_11 := x=2 & y=1 & v=1 & type=1 ;
DEFINE flag_st_12 := x=3 & y=1 & v=1 & type=1 ;
DEFINE flag_st_13 := x=4 & y=1 & v=1 & type=1 ;
DEFINE flag_st_14 := x=1 & y=2 & v=1 & type=1 ;
DEFINE flag_st_15 := x=4 & y=2 & v=1 & type=1 ;

--Strategy #1
DEFINE flag_full_stop := flag_st_2 | flag_st_3 | flag_st_4 | flag_st_5 |
flag_st_6 | flag_st_7 | flag_st_8 | flag_st_9 | flag_st_10 | flag_st_11 |
flag_st_12 | flag_st_13 | flag_st_14 | flag_st_15 ;
DEFINE flag_inhib_rot := flag_st_0 | flag_st_1 ;
```

The strategies generated by SMOF are hard to read whenever the number of warning states and/or interventions is high.

☞ The readability of the strategies is of low quality when there are more than a few warning states.

After a step of manual simplification, the strategy can be presented as follows:

```
--Strategy #1
DEFINE flag_full_stop := type=1 &
(x=1 | x=2 | x=3 | x=4) & (y=1 | y=2);
DEFINE flag_restrict_right_curve := type=0 & x=3 & (y=1 | y=2);
```

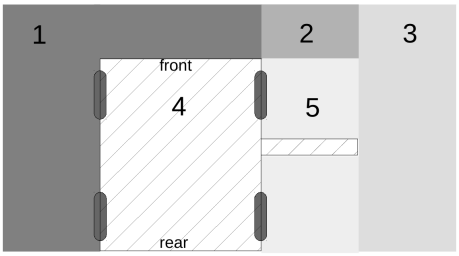

Figure 2.9: Obstacle occupation zones around the robot

It means that the robot will brake when a high obstacle is close, and restrict the curve when a small obstacle is in the deported sensor zone.

The monitor will therefore be intervening continuously when the robot is measuring lights, as they are considered as obstacles with respect to this invariant.

The Cartesian model is useful to understand the problem, but not completely realistic. Indeed, the robot can meet more than one obstacle at a time, and each obstacle has a spatial extension. For example it could be in the zones `x=2` and `x=3` at the same time. This is impossible to model with a single coordinate pair `(x,y)`.

To take into consideration multiple obstacles and their potential spatial extension, we chose to model occupation zones. Several zones can be occupied at the same time, accounting for either an obstacle spreading on several `(x, y)` coordinates (e.g., a wall, a large object) or for several separate obstacles.

☞ Several iterations can be needed to model a complex invariant and synthesize a strategy for it.

Zones model: Five zones are defined as in Figure 2.9. Their boundaries are based on the same thresholds as in the Cartesian model, with possibly some grouping of the coordinate areas. For example, the warning areas at the left and in front of the robot are grouped into the same zone: they are observed by the same sensor (L1 in Figure 2.7), and the previous analysis suggests that they call for similar safety rules.

Table 2.5 shows the partitioning of zone variables into occupation classes. Value `0` encodes an empty zone. The meaning of other values depends on whether or not the type of obstacles is observable in the corresponding zone. For example, variable z_1 is two-valued with `1` indicating the occupation by obstacles of an unobserved type. Variable z_2 is three-valued in order to distinguish occupation by at least one high obstacle (`=1`), and occupation by low obstacles only (`=2`). The partitioning of z_3 is specific: while the type of obstacles is observed in this zone, the variable is two-valued. The case of occupation by low obstacles only is safe for this zone, so we decided to put it in the same class as emptiness. In practice, note that the occupation cases are determined by combining the results of several sensors. Assuming that the implementation is like that of Figure 2.7, the occupation by low obstacles only would be determined by the fact that the 2D lidar on the platform's corner sees something in the zone, while the lidar on the deported extension (placed at a higher height) sees nothing.

Zone z_1: Front and left side of the robot	Empty	z_1=0
	Occupied by at least one obstacle (type unknown)	z_1=1
Zone z_2: Front right side of the platform	Empty	z_2=0
	Occupied by at least one high obstacle	z_2=1
	Occupied by low obstacles only	z_2=2
Zone z_3: Right side of the robot	Empty or occupied by low obstacles only	z_3=0
	Occupied by at least one high obstacle	z_3=1
Zone z_4: The platform itself	Empty	z_4=0
	Occupied by an obstacle (physical contact)	z_4=1
Zone z_5: Right side of the platform	Empty	z_5=0
	Occupied by at least one high obstacle	z_5=1
	Occupied by low obstacles only	z_5=2

Table 2.5: Partitioning of zone variables for the full case model

This encoding focuses on the occupation of zones, and misses information on their spatial adjacency. In the `(x,y)` model, adjacency was easily captured by the continuity constraints on `x` and `y`. The notion of warning state (i.e., the states from which a catastrophic state is reachable in one step) then naturally emerged, since the obstacle is always seen with coordinates in a close area before getting too close. In the new model, we have to introduce constraints to represent this. For example, a constraint expresses the fact that z_4 cannot be occupied if, at the previous step, it was empty and all its neighboring zones were empty as well. A similar constraint is given for z_5. The stationary assumption for low obstacles is also quite difficult to model. In particular, we have to capture the fact that, if the robot is stopped, low obstacles cannot appear and disappear in its neighborhood. It is done by introducing specific auxiliary variables and constraints. For example, we detect a situation in which the robot stops with low obstacles only in z_5, and constrain this zone to take only non-empty values as long as the robot remains at a standstill. The variables and their dependencies are declared as presented in Figure 2.10.

☞ The modeling of the variables dependencies can be very arduous. Several iterations may be necessary.

The velocity variable remains the same as in the previous `(x,y)` model. We thereby define the catastrophic state as `cata := v=1 & (`z_4`=1 |` z_5`=1)`: the robot is moving and an obstacle is in the zone z_4 or z_5 (for high obstacles only).

The two available interventions are the same as before: the full stop and the restriction of moves towards the right side.

```
MODULE Memo(set, reset)
VAR m :  boolean;
ASSIGN
  init(m):=set;
  next(m):=case
        next(set)=TRUE : TRUE;
        next(reset)=TRUE : FALSE;
        TRUE : m;
        esac;

MODULE main
⋮
--Variable dependencies
--The low obstacles are standstill
--The low obstacles do not disappear when the robot is not moving
DEFINE set_empty_prohibited_z2:=(v=0 & z2=2);
DEFINE set_empty_prohibited_z5:=(v=0 & z5=2);
DEFINE reset_empty_prohibited:=v=1;
VAR empty_prohibited_z2 :  Memo(set_empty_prohibited_z2, reset_empty_prohibited);
TRANS empty_prohibited_z2.m=TRUE -> next(z2)!=0;
VAR empty_prohibited_z5 :  Memo(set_empty_prohibited_z5, reset_empty_prohibited);
TRANS empty_prohibited_z5.m=TRUE -> next(z5)!=0;
--The low obstacles do not appear when the robot is not moving
DEFINE set_low_prohibited_z2:=(v=0 & z2=0);
DEFINE set_low_prohibited_z5:=(v=0 & z5=0);
DEFINE reset_low_prohibited:=v=1;
VAR low_prohibited_z2 :  Memo(set_low_prohibited_z2, reset_low_prohibited);
TRANS low_prohibited_z2.m=TRUE -> next(z2)!=2;
VAR low_prohibited_z5 :  Memo(set_low_prohibited_z5, reset_low_prohibited);
TRANS low_prohibited_z5.m=TRUE -> next(z5)!=2;
-- The obstacles do not come from nowhere
TRANS z1 = 0 & z2 = 0 & z4 = 0 & z5 = 0 -> next(z4) = 0;
TRANS z1 = 0 & z2 = 0 & z3 = 0 & z4 = 0 & z5 = 0 -> next(z5) = 0;
--The low obstacles do not become high
TRANS z1=0 & z2!=1 & z3=0 & z4=0 & z5!=1 -> next(z5)!=1;
```

Figure 2.10: Declaration of the zones variables and their dependencies

The SMOF tool identifies 94 warning states before invariant violation (vs. 16 previously). They account for the combinations of all occupation cases of the zones.

The search space of candidate strategies is very large, and the rule synthesis tool does not succeed in computing the set of solutions in a realistic amount of time when using the variant returning a superset of the minimal strategies. The tool also has a fast exploration mode at the expense of potentially skipping solutions (the third variant of the algorithm, returning a subset of minimal strategies). In the case of this model, the fast exploration manages to proceed the model in 4 minutes but finds no solution. Indeed, this variant of the algorithm is too drastic specifically when sequential preconditions are used, which is the case here for the full stop intervention.

☞ The state space is too large for the tool to deal with. No solution can be automatically found.

Note that the number of warning states for this safety invariant (94) is way higher than anything that we had seen in previous case studies. The SMOF tool was not designed to handle such a large state space, and it performs well for other examples with a smaller amount of states.

We then manually encoded the strategy suggested by the analysis of the Cartesian model: full stop when a high obstacle gets too close to the robot, i.e., enters one of the zones, and restrict curve when a low obstacle is in front, behind or below the sensor support, i.e., in the zone 2 or 5.

```
-- Strategy #1
DEFINE flag_full_stop := z1=1 | z2=1 | z3=1 | z4=1 | z5=1;
DEFINE flag_restrict_right_curve := z2=2 | z5=2;
```

NuSMV confirms that this strategy is safe and universally permissive. Note that there are now states for which both interventions are active at the same time, e.g., when both z_1=1 and z_2=2. This is indeed necessary because the full stop alone does not prevent from colliding with a low obstacle in z_2, which can be closer than the braking distance.

2.1.5 Rules Consistency

Within SMOF approach, each SI is modeled separately and has a dedicated safety strategy. To check the effect of merging the strategies retained for each invariant, the SMOF models are turned into NuSMV modules and gathered in one global model. A main module contains the glue information.

First, the variables from different SMOF models may be dependent. The user has to add the dependencies, in the same way she entered them in SMOF models. A particular case of dependency is to use the same observation in two SMOF models. For example, consider the invariant SI_1 (the robot must not go too fast) of velocity limitation, that uses the variable $v_{SI_1} \in \{0,1,2\}$ and the invariant SI_4 (the robot must not collide with an obstacle) using the information of whether or not the system is at stop (variable $v_{SI_4} \in \{0,1\}$). These invariants share a common observation, the velocity, but they have

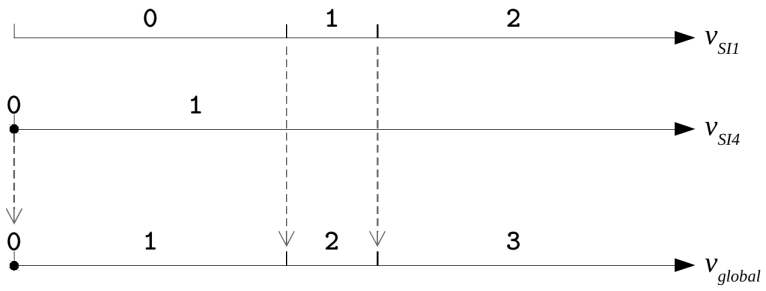

Figure 2.11: Merging partitions into a global variable

a different partition of its values. The two partitions are merged, and the result of the merging defines a global variable v_{global} (see Figure 2.11).

The evolution of these variables are constrained through invariant constraints:

```
--Declare the global variable
--Continuity(value_min, value_max, value_init)
DEFINE v_global:  Continuity(0,3,0);
--Declare the dependencies
INVAR v_global=0 <-> v_SI1=0 & v_SI4=0 ;
INVAR v_global=1 <-> v_SI1=0 & v_SI4=1 ;
INVAR v_global=2 <-> v_SI1=1 & v_SI4=1 ;
INVAR v_global=3 <-> v_SI1=2 & v_SI4=1 ;
```

Second, the launching of interventions has an impact on the whole system. Each time an intervention is asked by one local strategy, it may have an effect in all the SMOF models in which the intervention is modeled. The main module controls this by means of a global flag, defined as the disjunction of the local flags for this intervention. When the global flag is true, the effect of the intervention is determined in each SMOF model, under the local preconditions. Analysis of consistency aims to check whether two strategies, synthesized from different invariants, apply incompatible interventions at the same time (e.g., braking and accelerating). The model-checker easily detects roughly inconsistent cases like both increasing and decreasing a variable. But there might be less obvious cases not captured in the abstract models. So, we require the user to specify the forbidden combinations. Given a pair of incompatible interventions (i, j), their nonconcomitance is formulated as:

$$AG(\neg globalFlagInterv_i \wedge globalFlagInterv_j)$$

Permissiveness is also rechecked. An intervention launched by one SMOF model could impair the reachability of states in other SMOF models.

For the four invariants of the Sterela Case study, the consistency analysis was successful: the synthesized strategies are not conflicting with each other.

Figure 2.12: Manipulator robot from Kuka

2.2 Feedback from Kuka Case Study

The case study from Sterela gave us highlights of interesting points to develop. In this Section we revisit the results of a previous case study from Kuka in order to have further insight on them. We see that we also find the problems encountered during the Sterela case study. This case study was anterior to this work and detailed in [Machin, 2015].

2.2.1 System Overview

The studied robot is a manufacturing robot, that is meant to evolve in a factory in a workspace shared with human workers. It is composed of a mobile platform and an articulated arm (see Figure 2.12).

Its purpose is to pick up objects using its arm and to transport them. To do so, the arm can rotate and a gripper at its end can open and close to hold objects. Some areas of the workspace are prohibited to the robot. The hazard analysis has been performed on this robot in a previous project [SAPHARI, 2018] and a list of 13 safety invariants was identified. Among them, 7 could be handled by the monitor (presented in [Machin et al., 2018]). We review here two of them:

SI_I *the arm must not be extended when the platform moves at a speed higher than* $speed_{max}$*;*

SI_{II} *a gripped box must not be tilted more than* α_0.

Several formalizations have been explored for this case study, to explore the impact of considering different sensors and actuators, i.e., different observations and interventions. We present here one formalization for each invariant.

2.2.2 SI_I: Arm Extension with Moving Platform

We consider the invariant SI_I *the arm must not be extended when the platform moves with a speed higher than* $speed_{max}$. The available observations are s, the speed of the

Speed of the platform	**Real speed interval**	**Discrete variable**
Low	$s < speed_{max} - margin$	`s=0`
Within the margin	$speed_{max} - margin \leq s < speed_{max}$	`s=1`
Higher than the maximum allowed value	$s \geq speed_{max}$	`s=2`

Position of the arm	**Discrete variable**
Not extended beyond the platform	`a=0`
Extended beyond the platform	`a=1`

Table 2.6: Partitioning of the variables s and a

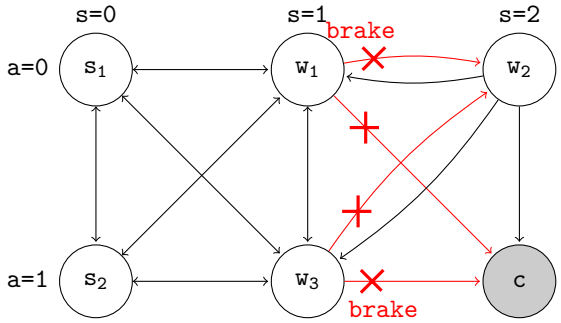

Figure 2.13: Strategy for the invariant SI_I with the braking intervention only

platform and a, the position of the arm. The observations are partitioned as detailed in Table 2.6. Considering the discrete representation of the variables, the catastrophic state can be expressed as `cata := s=2 & a=1` (high speed with extended arm).

Let us consider that one intervention is available: the brakes can be triggered and affect the speed. It can be applied at any time, i.e., it has no static precondition. But it will only be efficient (prevent the reachability of $s \geq speed_{max}$) if it is engaged when the speed threshold $speed_{max} - margin$ has just been crossed, i.e., `s=0` in the previous state: its sequential precondition is `s=0`. Indeed, the size of the margin is chosen precisely to have time to brake before reaching the undesired value. It is encoded as follows:

```
--Interv(stat_precond, seq_precond, effect)
VAR brake:  Interv(TRUE, s=0, next(s)!=s+1);
```

With this interventions, no strategy can be synthesized. In order to prevent the reachability of the catastrophic state, the brakes should be applied whenever the speed crosses the margin threshold, as represented in Figure 2.13.

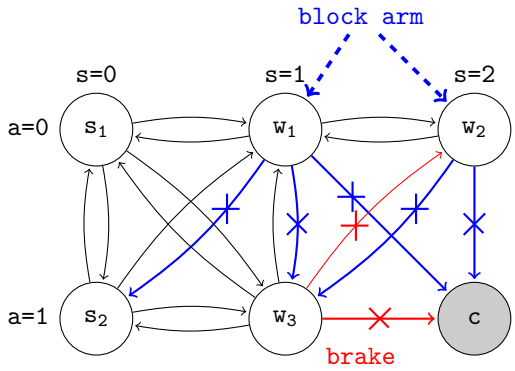

Figure 2.14: Strategy for the invariant SI_I with the two interventions

However, doing so prevents the reachability of any speed value above $speed_{max}$, even with the arm folded (state w_2 in Figure 2.13). The permissiveness property is violated. Here, the state machine can manually be analyzed to identify that the problem comes from the non-reachability of w_2 if the brakes are triggered. However, when SMOF returns as a result "no possible strategy", there is no way to know what went wrong in the synthesis, especially what reachability property could not be satisfied, therefore violating the permissiveness. It difficult for the user to find what is missing for a solution to be found.

☞ No strategy is found. A manual analysis allows us to identify that the reachability of a state is discarded if the brakes are triggered.

To overcome this issue, another intervention is added. The choice of this new intervention is not trivial. Here, as we want to allow the speed to reach values higher than $speed_{max}$, we deduce that the new intervention needs to impact the arm extension. We thus make it possible to block the extension of the arm. The effect of this intervention is to block the extension of the arm (`next(a)!=1`) and it can only be applied if the arm is not already extended (its static precondition is `a=0`).

```
-- Interv(stat.  precond., seq.  precond., effect)
VAR block_arm:  Interv(a=0, TRUE, next(a)=0);
```

With the addition of this new intervention, a single strategy is synthesized, represented in Figure 2.14: the brakes are triggered when the speed is in the margin and the arm extended, and the arm is blocked if the platform is going faster than the margin with the arm folded.

☞ To satisfy the permissiveness property, an additional intervention has been designed.

Adding new interventions for the monitor can be quite expensive: it means adding new actuators, making the existing ones reliable, or implementing new reliable software

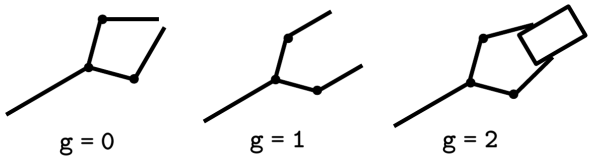

Figure 2.15: Positions of the gripper and corresponding discrete values

functions. This is similar to the Sterela collision invariant SI_4 where a new intervention had to be developed and implemented in the actual system, at a high integrity level.

Another solution would have been to make compromises on the reachability of states. Let us look again at the non-satisfying strategy (with regard to the permissiveness) that uses the braking intervention only (Figure 2.13). The warning state w_2 `:= s=2 & a=0` is not reachable. This state represents the robot overspeeding ($s > speed_{max}$) with the arm folded. Reaching this state might not be necessary for the robot to perform its task (move around the workspace carrying objects), and therefore during synthesis of a strategy, reachability of this state is not mandatory. It seems overly-stringent to require the reachability of this state, and more generally to require the universal reachability of every state. Some strategies discarding useless states could be satisfying.

☞ The permissiveness requirements can be overly-stringent, i.e., it might be acceptable that some states are not reachable.

2.2.3 SI_{II}: Tilted Gripped Box

The arm is able to pick boxes containing objects with its gripper. The boxes have an open top, therefore parts may fall if the box is tilted too much. We consider the invariant *SI_{II} a gripped box must not be tilted more than α_0*

To formalize this invariant, the required observations are the angle of rotation α of the gripper and the position of the gripper g. The angle is partitioned in three, according to the concept of margin: `α=0` represents a low rotation angle of the gripper, `α=1` an angle in the margin and `α=2` an angle higher than α_0. For the gripper, it can either be closed without a box, open or closed with a box (see Figure 2.15).

For the initial model presented in [Machin et al., 2018], the monitor could only brake the arm (prevent its rotation) and no control was possible on the gripper. No strategy was found. Indeed, the monitor would not be able to prevent the system from grabbing a box with the gripper already tilted more than α_0. The invariant was reformulated as *SI_{II}' : a gripped box must not be tilted more than α_0 if the robot is outside of the storage area.* The industrial partner indicated that objects falling (the box is tilted over α_0) in the storage area are not that dangerous as they would fall from a low height.

☞ Due to a lack of interventions, the safety requirement had to be weakened.

As an alternative solution to ensure SI_{II}, [Machin, 2015] chose to explore the effect of an additional intervention: the monitor can lock the gripper (prevent it from closing).

In this case, a single strategy is synthesized: prevent the rotation of the arm when a box is held and lock the gripper when the arm is tilted. No compromise on safety is made.

☞ An additional intervention has been designed. It allows the synthesis of a strategy satisfying the initial safety requirements.

2.3 Lessons Learned

In the previous two sections, we analyzed a number of invariants. As a reminder, the invariants from the STERELA case study are listed below:

SI_1 The robot must not exceed a maximum velocity v_{max};

SI_2 The robot must not move if the communication is not operational;

SI_3 The robot must not enter a prohibited zone;

SI_4 The robot must not collide with an obstacle.

The invariants reviewed from the Kuka case study are listed below:

SI_I The arm must not be extended when the platform moves at a speed higher than $speed_{max}$;

SI_{II} A gripped box must not be tilted more than α_0.

2.3.1 Encountered Problems

During the analysis of these 6 invariants, we faced several problems. We present below the problems encountered, and how they have been manually solved. The problems can be gathered in five main categories. Table 2.7 summarizes these different problems and their corresponding invariants, along with the solutions that have been chosen.

Understanding the failure of the synthesis: When SMOF fails to return a satisfying strategy with the initial requirements, as we saw for the invariants SI_3, SI_4, SI_I and SI_{II}, is it hard to assess if this is because the safety or permissiveness requirements are too stringent, or because of a lack of appropriate interventions. To identify if the safety requirements can be fulfilled, the user can remove completely the permissiveness requirements, and see if the synthesis returns a solution. If a strategy can be found, it means that safety is not the issue. Then, they have to think about whether the permissiveness requirements are too stringent, requiring the reachability of states that are not necessary to reach for the robot operation, or if the interventions cut too many transitions, discarding paths to useful states. For all but very small models, this analysis may be burdensome if not impossible. Some help to diagnose why the synthesis fails would be useful.

Problem	**Corresponding invariants**	**Explored solution**	**Desired contribution**
Understanding the failure of the synthesis	SI_3, SI_4, SI_I and SI_{II}	Synthesize a strategy without permissiveness requirements.	Help for the diagnosis.
No safe strategy found	SI_{II}	Weakening of the safety requirements; Addition of interventions.	Suggestion of interventions.
No permissive strategy found	SI_3, SI_4 and SI_I	Weakening of the permissiveness requirements; Addition of interventions; Addition of observations.	Definition of custom permissiveness; Suggestion of interventions.
State explosion	SI_4	Analysis on a simple model; Verify the manually implemented strategy	Optimization of the SMOF tool performances; Improvement of the pruning criteria.
Readability of the solutions	SI_4	Manual analysis and simplification.	Automatic simplification feature.

Table 2.7: Summary of the points of interest highlighted by the two case studies.

No safe strategy found: In some cases, SMOF was not able to find a safe strategy with the model proposed by the user, as with the invariant SI_{II}. For this invariant, no intervention was initially available to prevent the gripper from grabbing a box. The safety was weakened, only requiring the monitor to prevent tilting a box outside of a predefined storage area. Another solution was considered, making it possible not to weaken the safety requirements. A new intervention was defined — locking the gripper to prevent it from closing. This new intervention was designed after thoroughly analyzing the case, relying entirely on the user's expertise. Likewise, the Sterela case study required the proposal of a new intervention to restrict the moves in the direction of the lamps. Defining an appropriate intervention is not trivial, and some help from a tool would have been welcome.

No permissive strategy found: Another problem encountered is when SMOF cannot synthesize a permissive strategy. This was the case for the invariants SI_3, SI_4 and SI_I. For the collision invariant SI_4, an observation had to be added, making it possible to differentiate between types of obstacles. Thereby, the monitor could allow the lamps (low obstacles) to get close to the arm, but not other types of obstacles (high obstacles). For the prohibited zone invariant, the chosen solution was simply to require the simple permissiveness (reachability of all the states from the initial state), instead of the universal one (reachability of all the states from all the other states). This can easily be done with SMOF. However, it is hard to evaluate the impact of switching to simple permissiveness on the system's ability to perform its tasks. In the case of the collision invariant SI_4 and the arm extension invariant SI_I, switching to simple permissiveness was not sufficient to synthesize a strategy. In both these invariants, new interventions were added to allow a strategy to be synthesized. However, as noted above, designing new interventions can be arduous, and the help of a tool would have been welcome. For the arm extension invariant, another solution was considered, though not completely explored: requiring the reachability of some states only and discarding others. This solution would have meant that the strategy using only the braking intervention was satisfying.

Readability of the strategies: The strategies generated by SMOF are in the form of a list of definition of states (the warning states) and a list of triggering conditions for the interventions, expressed using the warning state definitions. When a large number of warning states exist, this can easily become unreadable. The user then has to manually simplify the triggering conditions of the interventions. A simplification feature would be very useful if not absolutely necessary.

State explosion: Finally, the last problem is the well-known one of state explosion. The branch-and-bound algorithm used for SMOF synthesis is not appropriate for large state spaces. To solve this problem, one would have to use a heuristic algorithm for instance. This problem was encountered for the invariant SI_4, which was the largest one we had to deal with so far. The temporary solution found is to manually implement the expected solution, and to verify it with the model checker. This meant going through several modeling iterations, beginning with a simplified version of the problem that lead

to understanding the realistic case. This problem is not solved in our work. We decided to focus on assisting the user in solving cases within the reach of SMOF, but where the synthesis fails to return a satisfying solution because there is none possible given the current model.

2.3.2 Implemented Solutions

We presented above the different kinds of problems that have been encountered while analyzing the two case studies. Several solutions have been used, and some of the solutions have been used several times in different contexts.

Diagnosing why the synthesis fails: When SMOF synthesis fails to return a satisfying solution with the initial requirements, it is hard to understand why. For the invariants presented in this chapter, the number of states was limited (except for the collision invariant SI_4), and we were able to manually analyze why the synthesis failed. For the prohibited zone invariant SI_3, and the arm extension invariant SI_I, when removing the permissiveness requirements, a safe strategy could be found. We then deduced that either the permissiveness was the problem, or the interventions. For SI_3, no other intervention was available, so we decided to change the permissiveness requirements. For SI_I, the two solutions were explored. Since this invariant only has six states, we could display the graph and use it to help us identify the impact of the safe strategy on the reachability of the states. For the collision invariant SI_4, a safe strategy could also be identified when removing the permissiveness requirements. For this invariant, no compromise could be made on the permissiveness (the robot needs to be able to reach states where it measures the lights). We thus decided to work on a new intervention. For the gripper invariant SI_{II}, with the first specification, no safe strategy could be synthesized. We then explored two solutions, accepting a weakening of the safety requirements, or adding an intervention to avoid weakening the safety requirements. There the model was also fairly simple, which allowed us to manually analyze it.

Weakening of the safety requirements: The safety requirements can be weakened. This means, defining new catastrophic states, that are a subset of the initial catastrophic states. Some of the initial catastrophic states can remain reachable. For instance, for the invariant SI_{II}, the safety has been weakened, allowing a box to be dropped when the robot is in the storage area. This is not a desirable solution, as it has a direct impact on the safety of the system as a whole. That could make the system not reliable enough for usage.

Addition of new interventions: The interventions cut paths between states: the paths to the catastrophic states, and others. There are two cases: they do not cut enough states, and paths remain to the catastrophic states, or they cut too many states, discarding all the paths to some non-catastrophic states. Changing or adding interventions can thus be a solution to ensure safety, and/or to restore permissiveness. For instance, the addition of an intervention locking the gripper made it possible to synthesize a safe and permissive strategy for the invariant SI_{II}. Defining interventions is challenging. It is arduous to identify which variable the intervention should control,

and how. For the two case studies presented earlier, it has been done by hand. The invariants were fairly simple and it was possible to identify the needed interventions for them. For more complex cases, it seems unrealistic to manually search for adequate interventions. For the collision invariant SI_4, designing the new intervention was not simple, and needed several iterations.

Weakening the permissiveness requirements: The permissiveness requirements, in their initial definition (universal reachability of all the non-catastrophic states) are very stringent. They do not necessarily correspond to any task that the system is meant to do. SMOF only has one way to lower the requirements on the permissiveness: switch to simple permissiveness. However, this is not necessarily appropriate. A better solution would be to have a finer tuning of the permissiveness requirement, specifying what states need to remain reachable. The others states could then be made unreachable by the synthesized strategy. This would allow the user to require the universal reachability of the important states (for the tasks of the robot), instead of having to lower the requirements for the states all together.

Addition of observations: The whole model depends on the observations chosen by the user. If the observations are not fine enough, or if no margin can be taken on them, it may be that the monitor has to trigger interventions early to prevent the reachability of the catastrophic states, and as a side effect prevent the reachability of other states that may be necessary to reach so that the robot can perform its task. Observations and the way they can be discretized are highly dependent on the system itself and the chosen technologies. It is not possible to define them automatically, and it is up to the user task to choose and define them.

Simplify the strategies: The strategies generated by SMOF can easily become unreadable. When they get too complex, the user may have to manually simplify them, which can be very burdensome. An automated simplification feature would be very helpful.

2.4 Conclusion

In this chapter, we applied SMOF to a new case study provided by the company STERELA. We also reviewed the results from two invariants extracted from a previous case study provided by the company KUKA. Doing so, we faced a certain number of problems. We saw that in some cases, SMOF cannot find any safe strategy, or can find a safe strategy but it is not permissive. We also saw that we reached the limits of the tool SMOF, with the largest model that we had to deal with since the tool has been implemented.

All these problems have been handled with manual procedures, relying on the simplicity of the models and on the user's expertise. It has been arduous, and will not be possible for larger and more complex models. A set of desirable contributions has been identified and is gathered in the last column of the Table 2.7.

Our work contributes in providing a set of solutions for the cases where SMOF indicated that there is no safe and permissive strategy: a diagnosis tool helps to identify why the synthesis fails to return a strategy, a customization template allows the tuning of the permissiveness requirements based on a set of essential functionalities to maintain and an interactive process suggests candidate safety interventions to inject into the synthesis process. Chapter 3 presents their specifications and explains how they fit together, in interaction with the user.

Take Aways:

- Evaluation of SMOF on a new case study;
- Need of tool optimization;
- Need of help for diagnosing why the synthesis fails;
- Need of a finer tuning of permissiveness;
- Need of help for the addition of interventions.

Chapter 3

Identified Problems and Overview of the Contributions

Contents

The SMOF process has been developed to automatically synthesize safety strategies for independent monitors. In the previous chapter, we applied the SMOF process to a new case study, a maintenance robot operating on airport runways. We also reviewed the results from a previous case study, a manufacturing robot working in a shared environment with humans. We could identify some needs to improve the SMOF process. Particularly, we are considering two recurring problems for the invariants presented in Chapter 2. The strategy synthesis does not always return a satisfying solution: in the first case, no safe strategy can be found, and in the second case, a safe strategy is found, but it does not satisfy the required permissiveness.

To solve these problems, several manual solutions have been used. The user could add some observations, add some interventions, weaken the safety requirements, or weaken the permissiveness requirements. However, all these solutions were burdensome to implement, as analyzing the problem can be arduous, and choosing the appropriate thing

to do is not trivial. It is also not clear how these solutions fit together, or if some of them could be automated.

In this chapter, we formalize the two identified problems, as well as the manually implemented solutions: when no safe strategy is found, or when the safe strategies are not permissive. From there, we identify which ones could be automated. This leads us to introduce our contributions that will be detailed further in the following chapters.

We detail and formalize the two identified problems in Section 3.1. We then formalize the solutions, and identify the needed contributions in Section 3.2. We give a high level view of our contributions in Section 3.3. We finally present our modifications to the SMOF process, integrating our contributions in Section 3.4, before concluding in Section 3.5.

3.1 Modeling of the Identified Problems

In this section, we detail how the invariant, the strategy, and the safety and permissiveness properties are modeled in the SMOF process. We also formally model the two cases where no satisfying strategy is found by SMOF: when no safe strategy exists, or when a safe strategy exists but no safe and permissive strategy can be found.

3.1.1 Invariant and Strategy Models

An invariant is modeled from a set of observable variables that allow the monitor to identify the state in which the system is. We call V the set of the n variables needed to formalize an invariant SI. Each variable v_i is defined on a definition domain D_i. The variables are partitioned according to the concept of margin (see Chapter 2). For example, a velocity variable vel is partitioned in three, the values above the catastrophic value ($vel > vel_{max}$), the values within the margin ($vel_{max} - margin < vel \leq vel_{max}$), and the low values ($vel \leq vel_{max} - margin$). P_i is a partition of the domain of definition D_i. The behavior of the system with respect to SI is then represented by the automaton $\mathcal{A} = (S_{all}, T, s_0)$, with:

- $S_{all} = P_1 \times \cdots \times P_n$ the set of states. We consider the set S of reachable states only (computed by SMOF), $S = S_s \cup S_w \cup S_c$ with S_s the safe states, S_w the warning states and S_c the catastrophic states;
- $T \subseteq S \times S$ the set of transitions. They represent the possible evolutions of the system;
- s_0 the initial state.

A set of interventions I is defined ($\mathcal{P}(I)$ being the power set of I). They are the possibilities for the monitor to control the system's behavior, impacting the variables in V.

A strategy R is defined as $R : S_w \to \mathcal{P}(I)$. It associates to warning states in S_w zero, one or several interventions in I. The strategy R modifies the behavior of the system.

The addition of interventions cuts transitions in T. It results in the modified automaton $\mathcal{A}_R = (S, T_R, s_0)$ where $T_R \subset T$.

3.1.2 Properties

We consider two types of properties. The safety property $Safe$ represents the violation of the invariant. A state violating the invariant satisfies the predicate *cata*. It is written in CTL as $Safe = AG(\neg cata)$. A strategy respecting $Safe$ is called a safe strategy.

$$\begin{gathered}\text{Let } R : S_w \to \mathcal{P}(I) \text{ be a strategy, let } \mathcal{A}_R \text{ be the modified automaton,} \\ R \text{ is safe iff } (\mathcal{A}_R, s_0) \vDash AG(\neg cata)\end{gathered} \tag{Safe}$$

It means that there is no path $\langle s \to \cdots \to s' \rangle$ in $\mathcal{A}_R$ where $s \in S \setminus S_c$ and $s' \in S_c$.

The set of permissiveness properties $Perm$ represent the reachability of the states. It ensures that the system can perform its tasks. The default permissiveness properties used are the universal reachability of every state: every non-catastrophic state is reachable from every other non-catastrophic state. A state is represented by its predicate giving the value of every state variable. For instance, the predicate of a state s representing a low velocity with the arm folded is $predicate(s) = (vel = low \wedge arm = folded)$. To simplify the readability of the formulas, we will slightly abuse notation and write s for both the state and its predicate.

The universal permissiveness property for a state $s \in S \setminus S_c$ is written in CTL as $AG(EF(s))$. A strategy respecting $Perm$ is called a permissive strategy (the empty strategy is by default permissive).

$$\begin{gathered}\text{Let } R : S_w \to \mathcal{P}(I) \text{ be a strategy, let } \mathcal{A}_R \text{ be the modified automaton,} \\ R \text{ is permissive iff for all } s \in S \setminus S_c, (\mathcal{A}_R, s_0) \vDash AG(EF(s))\end{gathered} \tag{Perm}$$

It means that there is a path $\langle s \to \cdots \to s' \rangle$ in $\mathcal{A}_R$ between every pair (s, s') of non-catastrophic states.

3.1.3 Identified Problems

As we saw in Chapter 2, a strategy satisfying $Safe$ and $Perm$ cannot always be found. Two cases exist (see left part of Figure 3.1).

No Safe Solution: The first case is when no strategy can be found that satisfies the safety property $Safe$. This means that for all the strategies, there exists at least one path to a catastrophic state, or equivalently, there is at least one reachable state that has a transition to a catastrophic state that is not cut by the strategy.

$$\begin{gathered}\text{For every strategy } R : S_w \to \mathcal{P}(I), \\ \mathcal{A}_R = (S, T_R, s_0) \text{ being the modified automaton,} \\ \text{there exists } s \in S_w, s' \in S_c, (s, s') \in T_R\end{gathered} \tag{No safe}$$

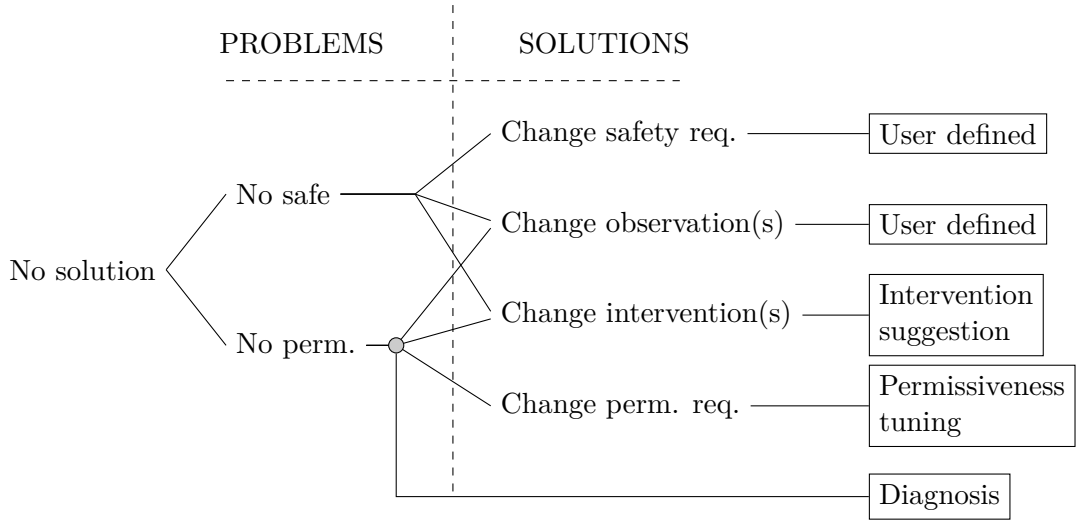

Figure 3.1: Identified problems, proposed solutions

If no safe solution is found, it can be for several reasons, as presented in Figure 3.1. The observations used for the modeling of the system's behavior may not be appropriate. Also, the safety requirement specified may be too stringent. Finally, the available interventions may not be appropriate, and may not cut enough (or the right) transitions, therefore not being able to discard all the paths to the catastrophic state.

The user can vary these three parameters (see Figure 3.1), and we formalize and discuss these possible variations in Section 3.2.

A Safe Solution, But Not Permissive: The second case is that a safe strategy can be found, but it does not satisfy $Perm$. This means that there is at least a pair of states between which there is no path:

$$\begin{aligned} &\text{There exists } R : S_w \to \mathcal{P}(I), (\mathcal{A}_R, s_0) \models Safe \\ &\qquad \text{and for all } R : S_w \to \mathcal{P}(I), \qquad \text{(Safe \& No perm.)} \\ &\text{there exists } s \in S \setminus S_c, (\mathcal{A}_R, s_0) \not\models AG(EF(s)) \end{aligned}$$

Similarly as presented in Figure 3.1, for the (No safe) issue, the impossibility to find a permissive strategy can be due to inappropriate observations, or interventions (interventions cutting too many transitions, discarding paths to certain non-catastrophic states). It can also be due to an overly-stringent permissiveness requirement. We explore in Section 3.2 several approaches to overcome these issues.

3.2 Manual Solutions to the Identified Problems

Several solutions to the identified problems have been manually implemented, as we saw in Chapter 2. In this section, we discuss and formalize these solutions, presented in the

right part of Figure 3.1, and identify which are the ones that could be (semi) automated, leading to our contributions.

3.2.1 Change the Observations

The observations, or observable variables are used to detect the violation of an invariant. The monitor follows their evolution, and when a defined threshold (potentially multi-observation) is crossed, it triggers the corresponding intervention(s) specified by the strategy. They are mostly from sensors, but can also be internal observations (status of a component, request, etc). If not enough observations are available, or the observations are not precise enough, it may not be possible to synthesize a strategy for the corresponding invariant. This was the case for the collision invariant SI_4, where an observation of the type of obstacle had to be added.

Changing the observations means having different sensing facilities to detect the violation of an invariant, and therefore potentially being able to trigger interventions only when they are strictly necessary (satisfying the permissiveness).

Formally, changing the observations means defining a new set of m variables V' (some of the variables can remain the same, i.e., $V \subset V'$ or $V \cap V' \neq \emptyset$). The new variables are defined on a new definition domain (D'_i for a variable v_i), resulting in different partitions (P'_i). This impacts the construction of the automaton modeling the system's behavior with regard to the considered invariant, $\mathcal{A}' = (S', T', s'_0)$. The new set of states is defined as $S' = P'_1 \times \cdots \times P'_m$.

Since the automaton is built from the available variables and their partitions, automatically inferring observations is not possible. An observation change implies a redefinition of the behavior model.

The user will have to specify the new variables and their partitions, and to model the catastrophic state(s) with these variables. This requires a specific knowledge of the system and the available technologies, since the observations are tight to the actual ways of observing the system's state and its environment.

There is no automatic solution for changing the observations. This task is left entirely to the user, relying on their expertise.

3.2.2 Change the Interventions

The interventions are ways provided to the monitor to affect the system's behavior. They are usually dedicated actuators or specific safety software functions. When the available interventions are not efficient enough —not cutting enough transitions, therefore not being able to discard paths to the catastrophic states— or not appropriate —not impacting the right variables— it will not be possible to synthesize a safe strategy. Also, if the interventions are too restrictive —cutting too many transitions, therefore discarding paths to non-catastrophic states— it may not be possible to satisfy the permissiveness properties.

Formally, changing interventions means defining a new set I' of interventions (some of the interventions can remain unchanged, i.e., $I \subset I'$ or $I \cap I' \neq \emptyset$). With this new set of interventions, new strategies $R : S_w \to \mathcal{P}(I')$ may exist.

The interventions affect the evolutions of the observations: they cut transitions to some states. It is possible to automatically identify what transitions need to be cut by the strategy. However, automatically generated interventions may not be implementable. The implementation of the interventions relies of the available technologies. An interactive approach, combining the automatic generation of guidelines for the design of new intervention and manual adjustments of the system characteristics by the user seems appropriate, and will be introduced in Section 3.3.3.

3.2.3 Change the Safety Requirements

The safety property specifies the part of the state space that cannot be reached without irreparable damages. As the considered systems are highly critical, compromising safety is usually not desirable. However, in some cases the invariant defined initially can be acceptably weakened. This was the case in the gripper invariant $\mathrm{SI_{II}}$ where dropping a box was considered acceptable in a defined storage area.

Weakening a safety requirement can formally be defined two ways. First, some states can be removed from the set of catastrophic states. The initial set of catastrophic state is partitioned in two, one part that remains catastrophic and one part with non-catastrophic states: $S_c = S'_c \cup S'_{nc}$. S'_c is the new set of catastrophic states. The states from S_{nc} are either safe or warning, resulting in new set of safe and warning states S'_s and S'_w: $S = S'_s \cup S'_w \cup S'_c$. This doesn't change the definition of the states themselves, i.e., the automaton. Only the *cata* predicate is different.

Second, the partition of variables values can be revised, or some variables added, yielding a new definition of states. For instance, an invariant specifying that the velocity must not exceed $10km/h$ could be changed to only prohibit a velocity value of $12km/h$ in a restricted area. This comes down to introducing new variables (a *restricted_area* variable for the previous example) and to drawing a new partition of some of the variables (the *vel* variable), i.e., to define a new set of variables V', thus a new set of states $S' = P'_1 \times \cdots \times P'_m$, where P'_i is the part defined for the variable v'_i.

Changing a safety invariant has to be done by experts with care, and cannot rely on an automated process. It impacts the safety of the whole system. Therefore, we leave this to the appreciation of the user.

3.2.4 Change the Permissiveness Requirements

Weakening the permissiveness requirements is preferred to compromising safety in most cases. The default permissiveness properties adopted by SMOF require the reachability of every non-catastrophic state from every other non-catastrophic state. In other words, every state that is physically possible to reach must remain reachable. This is a very stringent requirement as a lot of physically reachable states are not necessary for the

robot to perform its tasks. Changing the permissiveness requirement thus can be a good idea, and has a moderate impact on the robot's ability to function.

Formally, in our case, changing the permissiveness requirements means that we do not require the universal reachability of some states anymore. We define a new set of permissiveness properties $Perm' \subset Perm$. With this new requirement, some new strategies can be synthesized. In the resulting automaton, it is possible that no path exists between some pairs of states. Some states may be completely discarded.

In SMOF, there is no way to require the reachability of a subset of states, except by manually editing the properties generated by default, which is cumbersome. Still, a fine tuning of the permissiveness requirement would be interesting and allow more flexibility for the synthesis. If this cannot be fully automated in the choice of the states of interest, a template can be used for an easier specification of properties. We discuss this topic in Section 3.3.2.

3.3 High Level View of the Contributions

The experimental solutions described in Section 3.2 helped us identifying the needed automated features. The main identified needed contributions are the suggestion of intervention, and the tuning of permissiveness. The suggestion of interventions can be used whether the problem (No safe) or (Safe & No perm.) is faced. The tuning of permissiveness only is useful in the case of the (Safe & No perm.) problem.

In the case of (Safe & No perm.), the user will be able to choose between two automated solutions. However, SMOF does not give any precise detail regarding the situation, and it may be hard for the user to choose the appropriate solution. We propose to design a diagnosis module, that will analyze the problem, and give the user useful pieces of information about the non-satisfied properties, and the impact of the provided interventions on the reachability of states. This module will be introduced in Section 3.3.1. We then present the tuning of permissiveness in Section 3.3.2 and the intervention suggestion module in Section 3.3.3.

3.3.1 Diagnosis

When a safe strategy is found, but no permissive strategy exists, it may be complex to identify why. The initial model of the invariant, i.e., the possible physical behavior of the system, satisfies all the permissiveness properties: the system can move freely between states, all the states are universally reachable. When synthesizing a strategy, the addition of interventions to warning states will cut some transitions, to the catastrophic states, but also between non-catastrophic states. The interventions are not ideal, meaning that they do not only cut the transitions to catastrophic states, but also others, sometimes cutting too many transitions. Some paths to certain states can be completely discarded. Therefore, the corresponding permissiveness properties are violated.

Two main aspects are interesting to evaluate when considering a non-permissive strategy. First, what are the permissiveness properties that it does not satisfy? We

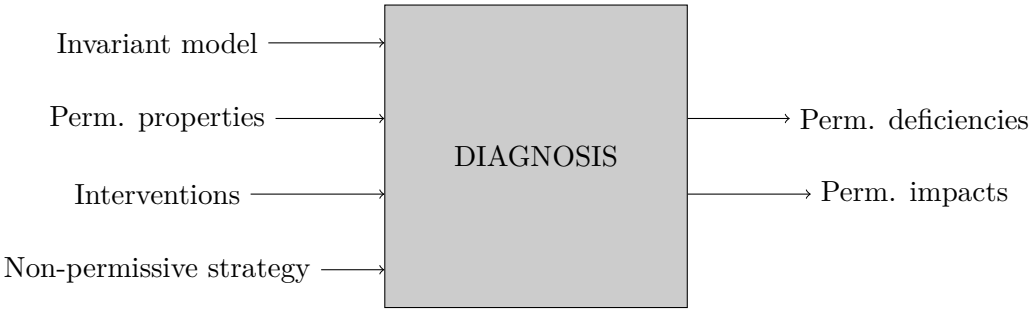

Figure 3.2: High level view of the diagnosis module

call the list of non-satisfied properties the *permissiveness deficiencies* of a strategy. The default permissiveness properties for SMOF are a list of CTL properties, specifying the universal reachability of every non-catastrophic state. A check of every property individually can provide the information of the specific reachability of every state.

Then, once we know which states are made unreachable by the strategy, we want to identify precisely which intervention, when triggered in a specific warning state, cuts the paths to the considered state. There can be several. We call this information the *permissiveness impact* of a couple (state, intervention(s)).

Providing these pieces of information to the user can help her make decisions on whether the loss of permissiveness is acceptable (i.e., the permissiveness requirements can be changed), and if not, what would be the appropriate interventions to replace. For this, we designed a diagnosis module (see Figure 3.2).

Inputs: The diagnosis module analyzes the impact of a non-permissive strategy on the system's behavior. Its inputs are the non-permissive strategy that is considered, and to evaluate it, the invariant model, that represents the behavior of the system with regard to the considered invariant; the permissiveness properties, in order to identify which one are satisfied or not, and the interventions, to check if their use impacts the reachability of some states.

Outputs: The two outputs are what we called the permissiveness deficiencies of a strategy, and the permissiveness impacts of the pairs (state, intervention(s)). The first one is the list of non-satisfied permissiveness properties. The second is the list of combinations of warning states and interventions from which there is no path to a specific state.

3.3.2 Tuning the Permissiveness

The generic permissiveness properties initially used by SMOF are very stringent: they require the universal reachability of all the non-catastrophic states. Reducing the permissiveness requirements to adapt them to the actual functionalities of the system can be a relevant approach. We propose a template to help the user defining custom permissiveness properties, as presented in Figure 3.3.

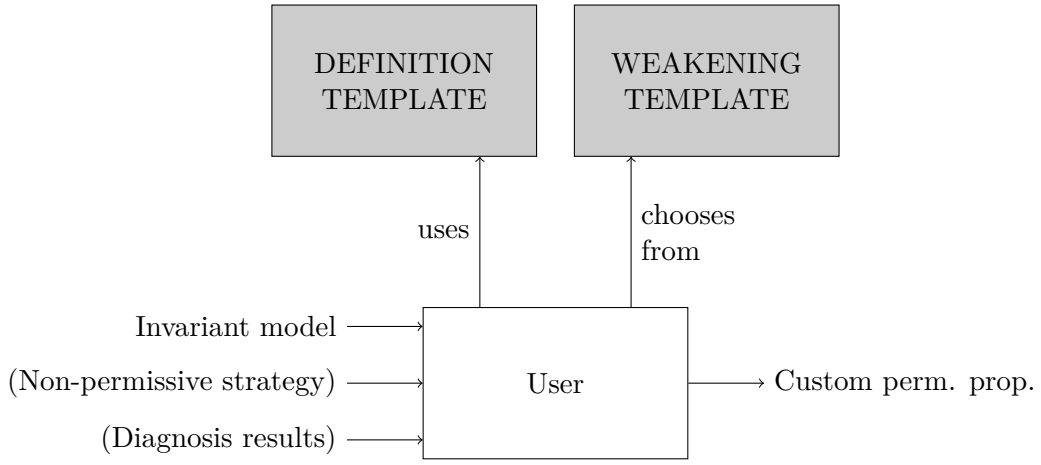

Figure 3.3: High level view of the tuning of the permissiveness properties

Definition Template: We consider that for the system to perform its tasks, it needs to be able to reach the corresponding states. For instance, for a "movement" task, the robot needs to reach states where the velocity is positive. The reachability properties are expressed in CTL for SMOF. However, the user may not be familiar with CTL, therefore we want to avoid their direct manipulation of formulas. We propose a simple template that the user can use to express the relevant functionalities. The user only needs to specify what set of states corresponds to the chosen functionality (e.g., $speed > 0$ for a "movement" functionality).

Weakening Template: In some cases, no satisfying strategy can be synthesized with the specified permissiveness properties, even if the user defined custom ones. They may have to weaken the permissiveness properties, allowing the monitor to have an impact on some of the functionalities. We propose ways to weaken the permissiveness, that represent different restrictions of the corresponding functionality. The user may choose among them.

Inputs: The user needs to define the permissiveness for one invariant at a time: it adapts it to the relevant functionalities with regard to a specific invariant, therefore needs to consider the model of the invariant, with the variables used for its definition. These variables must guide them for the identification of the relevant functionalities. For instance, the user does not need to define a "platform movement" functionality for an invariant only dealing with the position of an arm. Then, it happens that the user could not synthesize a permissive strategy. In this case, they may want to analyze the diagnosis results, to identify the blocking points with regard to permissiveness.

Output: Using the definition template, and potentially weakening the properties, results in a list of custom permissiveness properties. These properties are used as a replacement for the generic permissiveness properties for SMOF synthesis.

Figure 3.4: High level view of the suggestion of interventions

3.3.3 Suggestion of Interventions

Changing the available interventions can be an appropriate choice to make it possible to synthesize a satisfying safe and permissive strategy. We propose a semi-automated module that results in a list of suggested interventions. The view of this module is presented in Figure 3.4. The goal of this module it to create new interventions, that are appropriate to replace or be used along some of the interventions initially defined by the user, in a strategy respecting both the safety and permissiveness properties. Of course, there is no guarantee that the suggested interventions are implementable. However, it gives the user a precise idea of which kind of new interventions would be worth considering.

Inputs: The interventions suggestion modules has four inputs, among which two are optional. The first input is the model of the invariant, i.e., the system's behavior with respect to a specific invariant. It gathers the definitions of the variables, and how they evolve. It also encompasses the definition of the safety property. This will allow the suggestion module to identify which variables the new interventions should impact, and how. The second input is the list of interventions defined by the user. This will allow the suggestion module to take into account already existing interventions. The two following inputs are optional: an initial non-permissive strategy to improve and the diagnosis results for it. When a non-permissive strategy has been previously synthesized and the user wants to improve it by changing some of the triggered interventions, it is taken as a basis for the suggestion module. The diagnosis results need to be made available as well, and are used to remove the inappropriate interventions from the non-satisfying strategy. This step is optional. Using an existing strategy as a starting point for the suggestion module reduces the necessary exploration. Some interventions are already attached to some of the warning states: the suggestion module will have less states to explore, therefore will finish faster.

Output: The suggestion module results in a list of new interventions, that have been identified as good candidates for the synthesis of a safe and permissive strategy. Even though these interventions have been filtered so they are realistic, they still may not be implementable. They need to be reviewed by the user.

Interaction with the User: Defining suitable interventions cannot be done fully automatically. The interventions need to be implementable: the appropriate actuators or software functions need to exist or possibly be added. This depends on the technologies

used in the system, and on the available resources. It therefore requires validation from the user, to ensure that the suggested interventions are valid. We aim at limiting as much as possible the user's tasks, and the interventions will be presented for review only after a major step of sorting and simplification. The suggestion module presents a tentative list of interventions to the user. They can remove the interventions that they know won't be implementable, or modify the suggested ones (e.g., by adding preconditions). The modified list of interventions can then be used for the synthesis with SMOF.

3.4 Extension of SMOF Process and Modularity

In the previous sections, we presented the possible solutions when SMOF does not return a satisfying result. We introduced our three main contributions, that assist the user in solving this problem.

In this section, we detail the typical process that the user can follow when synthesizing safety strategies. We extend the SMOF process to integrate our contributions, and show how they can be used together. We also discuss the modularity of our approach. Our contributions can be viewed as a toolbox for the definition of monitors, and can be used interactively in different orders.

3.4.1 Typical Process

A global view of the process SMOF extended with the solutions detailed earlier is presented in Figures 3.5 and 3.6.

The first step is for the user to follow the SMOF process that has been presented in Chapter 1, and applied in Chapter 2 (see Figure 1.4): after extracting safety invariants from a hazard analysis, the user models them, specifying the available observations, interventions, and the desired safety properties. The permissiveness properties are the universal ones automatically added by SMOF. The SMOF synthesis is then launched. At the end of this step, two cases are possible. A safe and permissive strategy is found: this is the end of the process. The other case is the one we are interested in: no safe and permissive strategy is found.

From there, the first possibility is that no safe strategy is found (the problem (No safe), see Section 3.1). The user then has three choices: changing the observations, the interventions, or the safety requirements. This choice is left to their appreciation. If the user chooses to change the interventions, they can use the dedicated suggestion module that has been introduced in Section 3.3.3. After the change is made, the resulting new model of the observations, safety requirements, or interventions is used as an input for the SMOF process: the synthesis is launched again, resulting or not in a satisfying strategy.

The second possibility is that a safe strategy is found, but there exists no permissive strategy (the problem (Safe & No perm.), see Section 3.1). In this case, the user can use the diagnosis module presented in Section 3.3.1 to help them analyze the problem and choose the preferred solution. According to the results of the diagnosis, the user

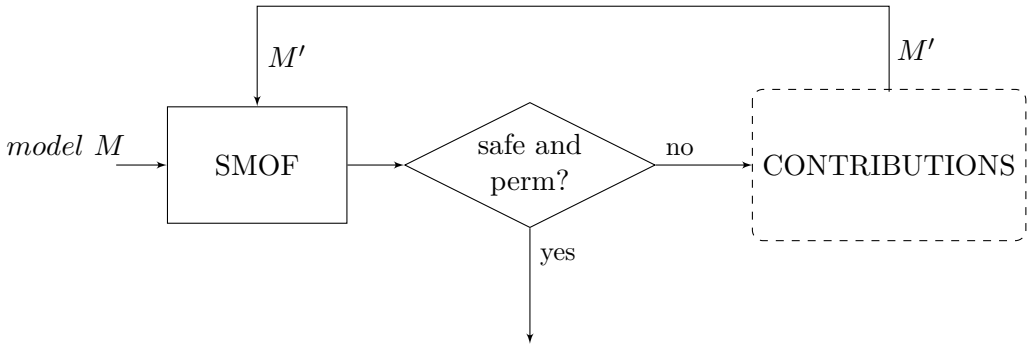

Figure 3.5: High level view of the SMOF V2 process

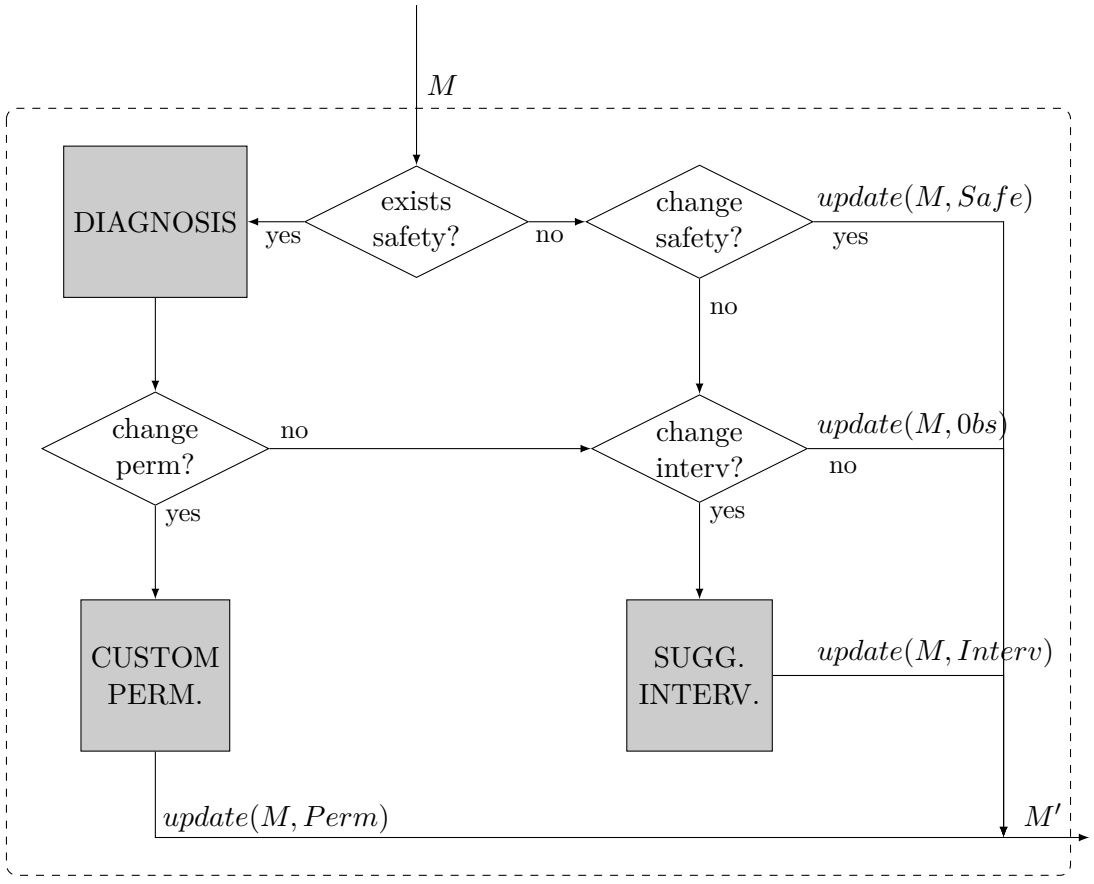

Figure 3.6: View of the contributions

can decide to change the observations, change the interventions (using the suggestion of intervention module), or change the permissiveness requirements (using the permissiveness tuning module). The new model of the observations, interventions or permissiveness requirements is then used for a new SMOF synthesis.

3.4.2 Flexibility

A typical process is presented above, but the user can deviate from it. Whatever the initial problem is, the mentioned solutions can be used together, in parallel. Several changes to the initial model may be needed. For instance, in the collision invariant SI_4, a new observation was first added. Then, an intervention was added, and only the combination of these two changes could allow a satisfying strategy to be found.

3.5 Conclusion

We saw in the previous chapter, when analyzing case studies, that when synthesizing a safety strategy with SMOF, it happens that no solution exists. We formalized the two cases that arise: when no safe strategy can be found, or when a safe strategy is found but no permissive strategy exists. These too problems have been manually solved in previous studies, by changing the observations or interventions, or reviewing the safety or permissiveness requirements. We discussed and formalized these solutions. We identified which solution need to be automated, to assist the user, and this lead us to specify three contributions. The first one is a diagnosis module, that helps the user analyzing the case and making a decision about the changes to make to the system model or the requirements. The second one is a template for the tuning of the permissiveness requirement. It allows the user to define custom permissiveness properties, adapted to the system required functionalities. The third one is a suggestion module for the definition of new interventions. It is an interactive module, where the interventions are inferred from the system model and reviewed by the user. These three contributions will be developed further in the next chapters, where we will also apply them to example and show the improvement compared to manually solving the identified problems. For the two first ones, we will present the tools that have been implemented to support the contributions. The third one is more prospective and has not yet been implemented: we will present the concepts definitions and the core algorithms.

Take Aways:

- Two problems are identified: no safe solution found, or a safe solution is found but no permissive solution exists;
- Four parameters can be modified to solve these problems: change the observations, the interventions, the safety requirements or the permissiveness requirements;
- We identified three needed contributions: a diagnosis module, a suggestion intervention module, and a permissiveness tuning module;

- They can be viewed as a toolbox to support the interactive search for a solution.

Chapter 4

Diagnosing the Permissiveness Deficiencies of a Strategy

Contents

The synthesis of strategies with SMOF may report that there is no safe and permissive solution. Two cases exist, as presented in Chapter 3: no safe strategy can be found, or a safe strategy is found but no safe and permissive strategy exists (i.e., the tool only finds non-permissive strategies). In this chapter we are interested in the latter case. In this case, the user can choose to change the permissiveness requirements, the interventions, or the observations. In order to be able to choose between these solutions, the user needs to know why the synthesis failed to return a permissive strategy.

The objective of this chapter is to provide a way to help the user identify the impact of a strategy on the permissiveness requirements (which we call the permissiveness deficiencies). This can help the user analyzing the failure of the synthesis to return a permissive strategy. We start with presenting a simplification feature using the solver z3 (Section 4.1) that we will use to increase the readability of our results. In Section 4.2, we develop and formalize the concepts of the permissiveness deficiencies of a strategy (the permissiveness properties that it does not satisfy), and of the permissiveness impact

of a couple (state, intervention(s)) (the states that are not reachable from it). We then present an algorithm to evaluate these two parameters, that includes a simplification step making the results readable (Section 4.3). We finally apply this on two examples, extracted from the presented case studies (Section 4.4), before concluding in Section 4.5.

4.1 Preliminaries - Readability of the Strategies

The readability is primordial for the user to be able to use the strategies, and for them to use any result provided by our toolset. In this section we propose a gateway with the z3 solver to take advantage of its simplification procedures. We show here how we use these simplifications for increasing the readability of the strategies. In the next section we will use the simplifications on the results of the diagnosis tool we will present.

4.1.1 Initial Display of the Strategies

When SMOF outputs a strategy, or a set of strategies, it is written as follows:

```
List of warning states;
⋮
List of strategies;
⋮
```

Let us consider for instance the invariant SI_4 (the robot must not collide with an obstacle), that is the most complex one we had to study in term of number of states. The list of warning states is as follows:

```
--Number of variables:6
--Number of warning states:94
--Number of interventions:1
DEFINE flag_st_0 := z1=1&z2=0&z3=0&z4=0&z5=0&v=0 ;
DEFINE flag_st_1 := z1=0&z2=1&z3=0&z4=0&z5=0&v=0 ;
DEFINE flag_st_2 := z1=1&z2=1&z3=0&z4=0&z5=0&v=0 ;
DEFINE flag_st_3 := z1=0&z2=2&z3=0&z4=0&z5=0&v=0 ;
⋮
DEFINE flag_st_91 := z1=1&z2=1&z3=1&z4=0&z5=2&v=1 ;
DEFINE flag_st_92 := z1=0&z2=2&z3=1&z4=0&z5=2&v=1 ;
DEFINE flag_st_93 := z1=1&z2=2&z3=1&z4=0&z5=2&v=1 ;
```

It is very hard and error-prone to read through the 94 warning states.

Below is the strategy synthesized for this invariant, with only the braking intervention (note that this strategy is not permissive):

```
DEFINE flag_brake := flag_st_0 | flag_st_1 | flag_st_2 | flag_st_3 | flag_st_4
| flag_st_5 | flag_st_6 | flag_st_7 | flag_st_8 | flag_st_9 | flag_st_10 |
flag_st_11 | flag_st_12 | flag_st_13 | flag_st_14 | flag_st_15 | flag_st_16 |
flag_st_17 | flag_st_18 | flag_st_19 | flag_st_20 | flag_st_21 | flag_st_22 |
flag_st_23 | flag_st_24 | flag_st_25 | flag_st_26 | flag_st_27 | flag_st_28 |
flag_st_29 | flag_st_30 | flag_st_31 | flag_st_32 | flag_st_33 | flag_st_34 |
flag_st_35 | flag_st_36 | flag_st_37 | flag_st_38 | flag_st_39 | flag_st_40 |
flag_st_41 | flag_st_42 | flag_st_43 | flag_st_44 | flag_st_45 | flag_st_46 |
flag_st_71 | flag_st_47 | flag_st_48 | flag_st_49 | flag_st_50 | flag_st_51 |
flag_st_52 | flag_st_53 | flag_st_54 | flag_st_55 | flag_st_56 | flag_st_57 |
flag_st_58 | flag_st_59 | flag_st_60 | flag_st_61 | flag_st_62 | flag_st_63 |
flag_st_64 | flag_st_65 | flag_st_66 | flag_st_67 | flag_st_68 | flag_st_69 |
flag_st_70 | flag_st_72 | flag_st_73 | flag_st_74 | flag_st_75 | flag_st_76 |
flag_st_77 | flag_st_78 | flag_st_79 | flag_st_80 | flag_st_81 ;
```

No need to say this is hardly usable.

We then propose to use the solver z3's simplification procedures to increase the readability of the strategies.

4.1.2 Simplifications with z3

The solver z3 offers several algorithms, that are called tactics, for the simplification of logical formulas. We chose to use ctx-solver-simplify. This tactic can be quite expensive as it uses a solver to check every parts of the formula to simplify. If a sub-formula is found not satisfiable, is can be replaced by false in the global formula, therefore making the simplification more efficient.

The solver z3 uses the SMT-LIB language [SMT-LIB, 2018], therefore we need to re-write our formulas in this language. We implemented a parser-rewriter that will translate smv models (used by SMOF) into smt models for z3 and reverse. The z3 models are as follows:

```
;for every variable:
(declare-datatypes () ((varTYPE varval0 ...  varvaln)))
(declare-fun var () varTYPE)
;for every intervention:  interv0:  inactive, interv1:  active
(declare-datatypes () ((intervTYPE interv0 interv1)))
(declare-fun interv () intervTYPE)
;formula to simplify:
(assert (formula))
(apply ctx-solver-simplify)
```

The simplification tactics may need to be run several times in order to get to the simplest possible formula.

The simplification algorithm that we designed, using z3, takes as input the model, the strategy file, containing the warning states definition and the interventions triggering condition definitions (the flags) and returns the simplified flag of the intervention.

For SI_4, the simplified strategy is:

```
DEFINE flag_brake:= (z4 = 1 & v = 0) | (z5 = 1 & v = 0) | (z5 = 2 & v = 0) |
(z1 = 1 & z4 = 0 & z5 = 0) | (z2 = 1 & z4 = 0 & z5 = 0) | (z2 = 2 & z4 = 0 &
z5 = 0) | (z3 = 1 & z4 = 0 & z5 = 0) ;
```

This is easier to read, as the formula has less clauses and one doesn't have to back and forth between the warning states definition and the intervention's flag definition. This can also be directly implementable: the triggering condition is directly written in a form using the variables values.

4.2 Concepts and Definitions

In this section, we review the definitions of a strategy, and of the permissiveness properties in SMOF. We analyze the problem identified in Chapter 3 when a safe strategy is found but no permissive strategy exists. We then present and formalize the concepts of the permissiveness deficiencies of a strategy, and of the permissiveness impact of a couple (state, intervention(s)). We explain how these concepts provide some visibility to the user on the impact of a non-permissive strategy on the system's ability to function.

4.2.1 Strategies and Permissiveness Properties

Let us consider an invariant SI. The behavior of the system with respect to SI is represented by the automaton $\mathcal{A} = (S_{all}, T, s_0)$, with:

- $S = S_s \cup S_w \cup S_c$ the set of reachable states (computed by SMOF, $S \subseteq S_{all}$), with S_s the safe states, S_w the warning states and S_c the catastrophic states;
- $T \subseteq S \times S$ the set of transitions. They represent the possible evolution of the system;
- s_0 the initial state.

The monitor has ways to control the system through a set of interventions I.

A strategy R for an invariant SI is a function that associates intervention(s) in $\mathcal{P}(I)$ (the power set of I) to warning states in S_w: $R : S_w \rightarrow \mathcal{P}(I)$. The association of interventions to warning states cuts transitions between states, thereby reducing the system's ability to freely evolve within the state space. It results in the modified automaton $\mathcal{A}_R = (S, T_R, s_0)$ where $T_R \subset T$.

Two types of properties are considered in SMOF. The safety property specifies the non-reachability of the catastrophic state(s). It is written in CTL as $AG(EF(\neg cata))$, i.e., all the reachable states are non-catastrophic states. The permissiveness properties

Let $R : S_w \to \mathcal{P}(I)$ be a strategy, let $\mathcal{A}_R$ be the modified automaton,
R is safe iff $(\mathcal{A}_R, s_0) \vDash AG(\neg cata)$,
R is permissive iff for all $s \in S \setminus S_c, (\mathcal{A}_R, s_0) \vDash AG(EF(s))$.

Figure 4.1: Definitions of safe and permissive strategy (repeated from Chapter 3)

There exists $R : S_w \to \mathcal{P}(I), (\mathcal{A}_R, s_0) \vDash AG(EF(\neg cata))$
and for all $R : S_w \to \mathcal{P}(I)$, there exists $s \in S \setminus S_c, (\mathcal{A}_R, s_0) \nvDash AG(EF(s))$

Figure 4.2: Definition of the Safe & No perm. problem (repeated from Chapter 3)

specify the reachability of the non-catastrophic states. They are intended to ensure that the system keeps its abilities.

In SMOF, the tasks are not specified in the invariant model, therefore we consider that to allow the system to perform its tasks, *all* the non-catastrophic states need to remain reachable, from every other non-catastrophic state. This type of permissiveness is called *universal permissiveness*. For a state $s \in S \setminus S_c$, it is written in CTL as $AG(EF(s))$. Note that we again slightly abuse notation, writing s both for the state and its characteristic predicate.

A strategy respecting the safety property is called a safe strategy, and a strategy respecting the universal permissiveness properties is called a permissive strategy (see Figure 4.1 for definitions). The goal of SMOF synthesis is to find a safe and universally permissive strategy. In most cases, finding a safe strategy is easy —shutting down the system is safe in most cases. Finding a permissive strategy is more complicated. Guaranteeing the permissiveness of a strategy is important as it ensures the good functioning of the system. However, the universal permissiveness properties may not always be satisfiable.

4.2.2 Problem: Safe & No perm.

When the SMOF synthesis is not able to return a safe and permissive strategy, we face the problem "Safe & No perm." that we formalized in Chapter 3 (see Figure 4.2 for a reminder): SMOF is not able to find a strategy that satisfies *all* the universal permissiveness requirements. For instance, a safe strategy that would satisfy the universal reachability of all the states *but one* would be discarded.

In SMOF, two options are available to weaken the permissiveness requirements to allow more flexibility for the synthesis. First, another type of permissiveness can be used to replace the universal permissiveness. We call it the *simple permissiveness*. The simple permissiveness specifies the reachability of every non-catastrophic state from the initial state only. It is written in CTL as $EF(s)$ for a state $s \in S \setminus S_c$. The simple permissiveness allows the monitor's intervention to be irreversible, i.e., some states can

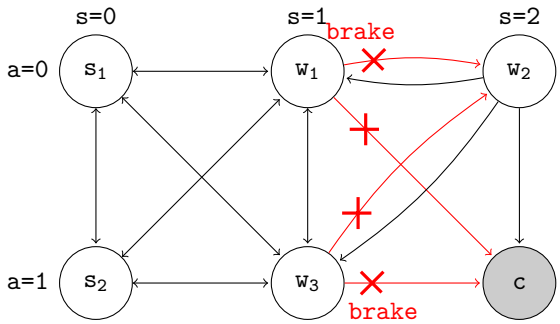

Figure 4.3: Strategy for the invariant SI_I with the braking intervention only (repeated from Figure 2.13)

be made unreachable after an intervention is triggered. This was the case for the invariant SI_3 (the robot must not enter a prohibited zone): after the full stop is triggered, the robot is blocked close to the prohibited zone and cannot move again. An intervention of the operator is necessary.

The second option, when the simple permissiveness is not sufficient for finding a safe and permissive strategy, is to use the safety only. The permissiveness requirements are removed, and the synthesis searches only for a safe strategy.

When one of these options is successfully used for the synthesis, SMOF returns a solution that satisfies *at least* the set of weakened properties (potentially empty). However, the exact set of properties that are satisfied by the strategy is unknown. There is no information given on the reachability of the states individually. Some states could remain universally reachable, when other would only be simply reachable, or not reachable at all. For instance, let us consider the strategy synthesized for the invariant SI_I (the arm must not be extended when the platform moves with a speed higher than $speed_{max}$) (see Figure 4.3 for a reminder). This strategy is not permissive: not all the permissiveness properties are satisfied. The state w_2 is not reachable. However, the states w_1 and w_3 are universally reachable: they can be reached from any other reachable non-catastrophic state.

SMOF does not provide an understanding of the reachability of the states individually. This would be a useful piece of information, since some states may not be necessary to reach to perform the system's tasks. In the example above (Figure 4.3), the state w_2, representing the robot overspeeding with the arm folded is not crucial for the functioning of the robot. The represented strategy, that prevents the reachability of this state, can therefore be acceptable, even though it is not fully permissive.

In the next section we propose two ways to analyze a non-permissive strategy, that allows for a better understanding of the reachability of the states individually.

4.2.3 Permissiveness Deficiencies of a Strategy

In the case of "Safe & No perm.", i.e., SMOF returns a list of safe but non-permissive strategies, the user only has the information that *not all* the permissiveness properties are satisfied by the strategies. They cannot identify precisely what are the permissiveness properties that are satisfied or not, i.e, the states that remain reachable or not. Therefore, the impact of the strategy on the system's ability to function (i.e., the reachability of states) is hard to evaluate.

To provide the user with some visibility on the impact of a strategy on the initial permissiveness requirements (the universal permissiveness), we propose to answer two questions. First, what are the permissiveness properties that are not satisfied by the strategy? Indeed, the only information provided by SMOF is if all the permissiveness properties are satisfied or not. Second, which intervention(s) triggered in which state prevents the reachability of the non-reachable states (the ones corresponding to the non-satisfied permissiveness properties)? The addition of interventions is the cause of the loss of permissiveness (the empty strategy is fully permissive). We want to identify which interventions have an impact on the permissiveness when they are triggered.

Let us take the abstract example in Figure 4.4. A strategy R is synthesized, satisfying the safety property but an unknown set of permissiveness properties. The first step is to identify what states are not reachable at all. These states can be identified by checking the CTL property $EF(s)$. This corresponds to the simple permissiveness, and the list of non-satisfied simple permissiveness properties is called the *simple permissiveness deficiencies* of a strategy. In the example, only the state s_n is not simply reachable. Analyzing which simple permissiveness property is not satisfied provides the information of the states that are not reachable, even from the initial state.

The second step is to identify, among the remaining reachable states, the ones that are not universally reachable (not reachable from *every* other non-catastrophic states). This is what we call the *universal permissiveness deficiencies* of the strategy R. It allows us to identify which states are reachable from the initial state but do not remain reachable after the intervention of the monitor. In the example, the properties $AG(EF(s_1))$ and $AG(EF(s_n))$ are not satisfied, i.e., the states s_1 and s_n are not universally reachable. Note that all the states that are not simply reachable are not universally reachable either, but we do not analyze them further for the universal reachability.

From there, for every non-satisfied universal permissiveness property, we want to identify the combination of (state, intervention(s)) from which the corresponding state is not reachable. For instance, consider the state s_1 in Figure 4.4, that has been diagnosed as not universally reachable. A second diagnosis step allows us to identify that there is no path from the combination (s_j, i_1) to s_1. This is what we call the *permissiveness impact* of (s_j, i_1). We denote with (s_j, i_1) the fact that the intervention i_j active in s_j, i.e., all the preconditions of i_j are satisfied, i_j is effective.

Permissiveness deficiencies of a strategy: Consider a set of permissiveness properties $Perm$. We call permissiveness deficiencies ($\mathcal{PD}$) of R the list of permissiveness

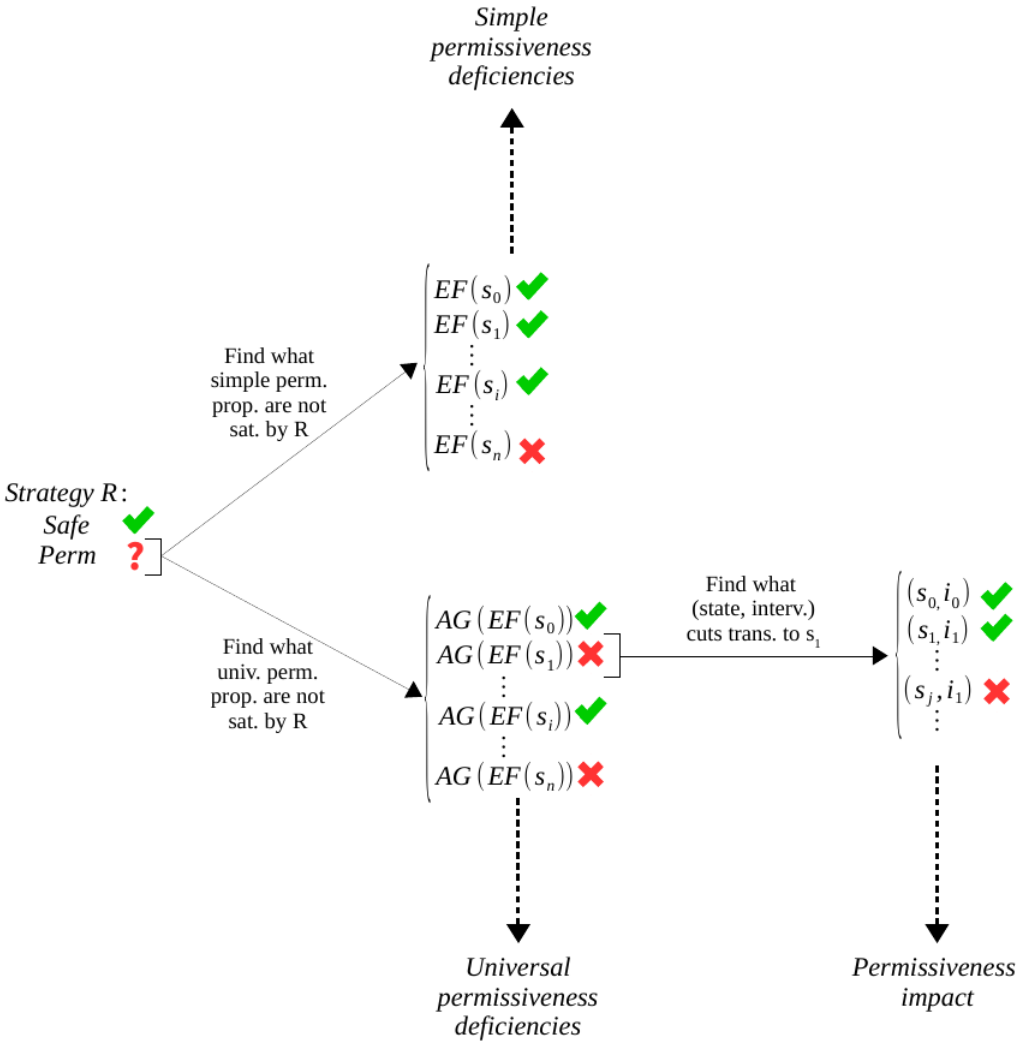

Figure 4.4: Principle of diagnosing the permissiveness deficiencies and impact of a strategy.

properties from $Perm$ that are not satisfied by R. $\mathcal{PD}$ makes explicit the permissiveness loss due to a strategy.

$\mathcal{PD}$ is split in two, $\mathcal{PD}_{univ}$, the list of non-satisfied universal permissiveness properties, and $\mathcal{PD}_{simple}$, the list of non-satisfied simple permissiveness properties. $\mathcal{PD}_{univ}$ allows the identification of the states that may not remain reachable after the intervention of the monitors. $\mathcal{PD}_{simple}$ denotes the states that are not reachable at all due to the strategy. If a state is not simply reachable, it is not universally reachable either: for every $s \in S \setminus S_c$ such as $EF(s) \in \mathcal{PD}_{simple}$, $AG(EF(s)) \in \mathcal{PD}_{univ}$.

Permissiveness impact of a couple (state, intervention(s)): We call permissiveness impact ($\mathcal{PI}$) of a couple (state, intervention(s)) the function $\mathcal{PI} : (S_w,\ \mathcal{P}(R(S_w))) \rightarrow S \setminus S_c$ that associates to a couple (state, intervention(s)) the list of states that cannot be reached from it (not considering the catastrophic states). S_w is the set of warning states. $R(S_w)$ is the set of interventions that are applied in the considered warning state ($\mathcal{P}(R(S_w))$ its power set, including zero intervention, and all the combinations of interventions), $R(S_w) \subseteq I$. $S \setminus S_c$ is the set of non-catastrophic states. It includes the safe and warning states.

To determine if a state s can be reached from the couple (s_w, i_w), we verify that if the state s_w with the intervention i_w active is reachable from the initial state, then from this combination (s_w, i_w), the state s is reachable. It corresponds to the following expression in CTL:

$$EF(s_w \wedge i_w) \implies EF(s_w \wedge i_w \wedge EX(EF(s))) \tag{4.1}$$

In the first part of this equation, $EF(s_w \wedge i_w)$, we state that the couple (s_w, i_w) can be reached from the initial state, i.e., s_w is reachable by a path such that the preconditions of i_w are satisfied. If that is the case, the second part of the equation is verified. The part $EX(EF(s))$ of the equation specifies that, after the next step, s will eventually be reached. Considering the next step is in particular necessary to check whether a state remains reachable from itself, even if the intervention enforces a change of state. In SI_3 (the robot must not enter a prohibited zone), the state `w1: d=1 & v=1` (the robot is moving close to the prohibited zone) is not reachable from itself, as the full stop intervention changes the value of v to zero and blocks it there.

We defined two concepts, the permissiveness deficiencies ($\mathcal{PD}$) of a strategy, and the permissiveness impact ($\mathcal{PI}$) of a couple (state, intervention(s)). In the following section, we detail the algorithm we follow to compute the $\mathcal{PD}$ and $\mathcal{PI}$ for a given non-permissive strategy.

4.3 Diagnosis Algorithm and Implementation

In this section, we detail the algorithms for the identification of the permissiveness deficiencies of a strategy, and of the permissiveness impact of a couple (state, intervention(s)). We then present some simplification features that help for the readability of

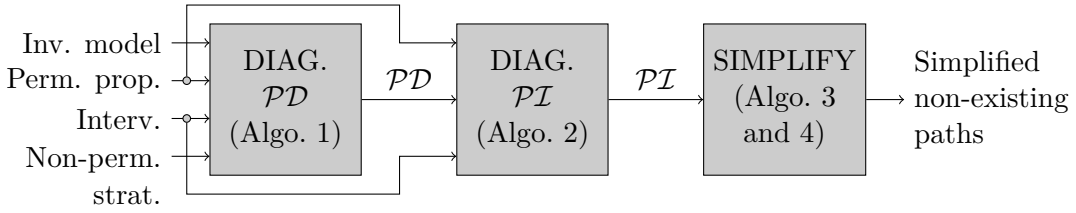

Figure 4.5: Process for use of the diagnosis and simplification algorithms

the results. The algorithms are used together as displayed in Figure 4.5. Finally we say a few words about the implementation of the diagnosis tool.

4.3.1 Algorithm

Identifying the permissiveness deficiencies of a strategy is the first step, and is relatively easy to do: the model modified by the addition of the strategy R (i.e., the automaton $\mathcal{A}_R = (S, T_R, s_0)$) is simply checked by a the model-checker NuSMV (for integration purposes with SMOF) for every permissiveness property *perm* individually, $Perm$ being the set of permissiveness properties ($Perm = Perm_{simple} + Perm_{univ}$, with $Perm_{simple}$ the set of simple permissiveness properties and $Perm_{univ}$ the set of universal permissiveness properties). The properties in $Perm_{simple}$ are expressed as $EF(s)$, for every $s \in S \backslash S_c$ (we use s for both the state and its characteristic predicate), and the properties in $Perm_{univ}$ as $AG(EF(s))$.

If the property is not satisfied, it joins the set $\mathcal{PD}(R)$ (the permissiveness deficiencies of R, i.e., the list of permissiveness properties that are not satisfied by R, $\mathcal{PD}(R)_{simple}$ for the simple permissiveness and $\mathcal{PD}(R)_{univ}$ for the universal permissiveness). Note that if a state is not simply reachable, it is also not universally reachable. We only check the universal reachability of the states that are simply reachable. The user can clearly evaluate if the loss of permissiveness for the strategy considered is acceptable in order to maintain safety. The corresponding algorithm is detailed in Algorithm 1.

The second step is to determine $\mathcal{PI}$ for every couple $(s_w, i_w) \in (S_w, R(S_w))$. We need to verify the CTL formula 4.1 for all of them. With the help of the model-checker, if a path exists to (s_w, i_w) (a path exists to s_w that triggers the intervention i_w in s_w), every path from (s_w, i_w) is explored until the state s is reached or the paths have exhaustively been checked. The corresponding algorithm is detailed in Algorithm 2. SMOF generates a variable that denotes the intervention activation: if the execution path satisfies the preconditions and the intervention is applied in the current state, the activation variable is true. In the algorithms, we abuse notation and write i_w for both the intervention name and its activation condition. The variable i_w is true if the intervention is active. Note that we check the impact of every intervention independently, and of the combinations of interventions. When a combination is checked, the activation conditions are conjoined.

For instance, let us consider the simple invariant SI_3 (the robot must not enter a prohibited zone). We have $\mathcal{V} = \{d, v\}$ the set of variables, where `d` $\in$ {`too close, close, far`}={`0,1,2`} and `v` $\in$ {`stopped, moving`}={`0,1`}. Only one in-

Algorithm 1: Identification of $\mathcal{PD}$ for a strategy R

Input: $Perm$, $\mathcal{A}_R = (S_{all}, T_R, s_0)$
for $perm_{simple} \in Perm_{simple}$ **do**
  result=Check($perm_{simple}$)
  if *result==false* **then**
    $\mathcal{PD}(R)_{simple} += perm_{simple}$
  end
end
for $perm_{univ} = AG(EF(s)) \in Perm_{univ}$ *with* $s \in S \setminus S_c$ *such as* $EF(s) \notin \mathcal{PD}(R)_{simple}$ **do**
  result=Check($perm_{univ}$)
  if *result==false* **then**
    $\mathcal{PD}(R)_{univ} += perm_{univ}$
  end
end
$\mathcal{PD}(R) = \mathcal{PD}(R)_{simple} + \mathcal{PD}(R)_{univ}$
Output: $\mathcal{PD}(R)$

Algorithm 2: Identification of the $\mathcal{PI}$ of the couples (state, intervention(s)) for a strategy R

Input: R, $\mathcal{A}_R = (S, T_R, s_0)$, $\mathcal{PD}(R)$
for $perm_{univ} = AG(EF(s)) \in \mathcal{PD}(R)$ **do**
  for $s_w \in S_w$ **do**
    for $i_w \in \mathcal{P}(R(s_w))$ **do**
      result=Check($EF(s_w \wedge i_w) \implies (EF(s_w \wedge i_w \wedge EX(EF(s))))$
      if *result==false* **then**
        $\mathcal{PI}(s_w, i_w) += s$
      end
    end
  end
end
Output: $\mathcal{PI}(s_w, i_w) \forall s_w \in S_w, i_w \in R(s_w)$

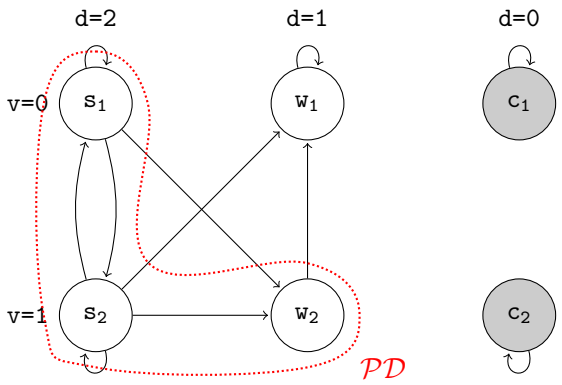

Figure 4.6: Permissiveness deficiencies for SI_3

tervention is available: `I={full stop}`, that stops the robot completely. The catastrophic state is defined as `cata: d=0`, i.e., `safe: d!=0` is the safety property.

Two warning states exist: w_1: `(d=1 & v=0)`, the robot is stopped close to the prohibited zone, and w_2: `(d=1 & v=1)`, the robot is moving close to the prohibited zone.

For this invariant, SMOF was only able to synthesize a non-permissive strategy (the simple permissiveness was used). It triggers the full stop in the two warning states:

$$R\texttt{(d=1 \& v=0)=\{full stop\}}$$
$$R\texttt{(d=1 \& v=1)=\{full stop\}}$$

The remaining possibilities of evolution for the system are represented in Figure 4.6. The transitions cut by the full stop intervention have been removed from the graph.

The algorithm 1 for identifying $\mathcal{PD}(R)$ results in the following:

$$\begin{aligned}\mathcal{PD}(R) = \{&\texttt{AG(EF(d=2 \& v=0))},\\ &\texttt{AG(EF(d=2 \& v=1))},\\ &\texttt{AG(EF(d=1 \& v=1))}\}\end{aligned}$$

This tells us that the strategy R does not satisfy the universal permissiveness for the states s_1: `d=2 & v=0`, s_2: `d=2 & v=1` and w_2: `d=1 & v=1`, i.e., when the robot is far from the prohibited zone (stopped or moving), and when the robot is moving close to the prohibited zone. These states can be reached from the initial state but not after the full stop is triggered. They are the states represented in the circle in Figure 4.6.

Then, we identify the permissiveness impact of each couple (state, intervention(s)) using the algorithm, Algorithm 2 (when `fullstop` is true it means that the full stop

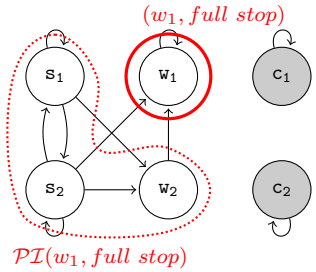

Figure 4.7: $\mathcal{PI}(w_1, full\ stop)$

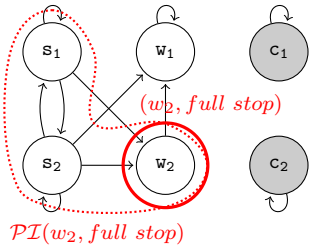

Figure 4.8: $\mathcal{PI}(w_2, full\ stop)$

intervention is active):

$$\begin{aligned}\mathcal{PI}(\texttt{d=1 \& v=0 \& full stop}) = \{&\texttt{d=2 \& v=0},\\ &\texttt{d=2 \& v=1},\\ &\texttt{d=1 \& v=1}\}\\ \mathcal{PI}(\texttt{d=1 \& v=1 \& full stop}) = \{&\texttt{d=2 \& v=0},\\ &\texttt{d=2 \& v=1},\\ &\texttt{d=1 \& v=1}\}\end{aligned}$$

This means that from the warning state $\texttt{w}_1$: d=1 & v=0 with the full stop active, the states $\texttt{s}_1$: d=2 & v=0, $\texttt{s}_2$: d=2 & v=1 and $\texttt{w}_2$: d=1 & v=1 are not reachable (see Figure 4.7). The same result is found for the state $\texttt{w}_2$: d=1 & v=1 (see Figure 4.8). This gives more information than $\mathcal{PD}(R)$ as it helps identifying that the full stop intervention in these two states contributes to the loss of permissiveness.

4.3.2 Simplification of the Diagnosis Results

The identified permissiveness deficiencies of a strategy, as well as the permissiveness impacts can easily become unreadable. If the result is too complex and hard to apprehend, it will very likely not be used. The whole point of the diagnosis we propose is to present a more readable list of permissiveness issues for a considered strategy, in order to help the user identify what changes are pertinent to apply to the permissiveness requirements, or to the observations or interventions they initially specified. We therefore propose a set of simplifications to improve readability. We reuse the gateway with z3 introduced in Section 4.1.

Several simplifications can be applied to help the user read the results of the diagnosis.

For the permissiveness deficiencies: The permissiveness deficiencies of the strategy can be gathered. All the states that are not universally reachable are gathered in a set of non-reachable states. Their characteristic predicates are simplified, resulting in a simplified characteristic predicate of the non-universally reachable set of state. Let $\mathcal{PD}(R)$ be the permissiveness deficiencies of the strategy R. Let $S_{\mathcal{PD}}$ be the set of states that are not universally reachable, i.e., for every $s \in S_{\mathcal{PD}}$, $AG(EF(s)) \in \mathcal{PD}(R)$. The

characteristic predicate of the set of states $S_{\mathcal{PD}}$ can be calculated by simplifying the expression $s_0 \vee s_1 \vee \cdots \vee s_i$, for $s_0, \cdots, s_i \in S_{\mathcal{PD}}$, with $i = Card(S_{\mathcal{PD}})$. This means that every state s in $S_{\mathcal{PD}}$ satisfies the simplified predicate $predicate(S_{\mathcal{PD}}) = s_0 \vee s_1 \vee \cdots \vee s_i$.

For SI_3 (the robot must not enter a prohibited zone), when analyzing the non-permissive strategy presented above (Section 4.3.1) the following permissiveness deficiencies were identified:

$$\begin{aligned}\mathcal{PD}(R) = \{&\texttt{AG(EF(d=2 \& v=0))},\\ &\texttt{AG(EF(d=2 \& v=1))},\\ &\texttt{AG(EF(d=1 \& v=1))}\}\end{aligned}$$

We simplify the characteristic predicates for the set of non-reachable states:

$$\begin{aligned}predicate(S_{\mathcal{PD}}) &= \texttt{(d=2 \& v=0) | (d=2 \& v=1) | (d=1 \& v=1)}\\ &= \texttt{d=2 | (d=1 \& v=1)}\end{aligned}$$

All the states that are not universally reachable ($\texttt{s}_1$, $\texttt{s}_2$, $\texttt{w}_2$), i.e., for which the universal permissiveness is not satisfied, satisfy the predicate `d=2 | (d=1 & v=1)`. This corresponds to the set of states in the dotted red circle in Figure 4.6.

For the permissiveness impact: The permissiveness impact of the couples (state, intervention(s)) shows from which state, triggering which intervention(s), a certain state is not reachable. If several $\mathcal{PI}$ are equal, it means some states are not reachable from several couples (state, intervention(s)). The characteristic predicates of these departure couples (state, intervention(s)) can be gathered (see Algorithm 3). Let us consider the abstract example in Figure 4.9. There is no path from the state s_0 with the intervention i_0 to the state s_2, and there is not path either from (s_1, i_1) to s_2. The characteristic predicates of (s_0, i_0) and (s_1, i_1) can be gathered. We compare all the permissiveness impacts (all the states that are not universally reachable) of all the couples (state, intervention(s)) (Algo. 3, line 2). If two permissiveness impacts are the same (Algo. 3, line 6), we gather the couples (state, intervention(s)) to compute the predicate representing the set of states that have the same permissiveness impact (Algo. 3, line 7).

We call departure predicate the simplified characteristic predicate of all the couples (state, intervention(s)) that have the same $\mathcal{PI}$, i.e., the departure predicate characterizes all the couples (state, intervention(s)) from which one state s is not reachable.

For the invariant SI_3, the couples $(\texttt{w}_1$, `full stop`$)$ and $(\texttt{w}_2$, `full stop`$)$ have the same permissiveness impact, $\mathcal{PI}(\texttt{w}_1$, `full stop`$)=\mathcal{PI}(\texttt{w}_2$, `full stop`$)=\{\texttt{s}_1, \texttt{s}_2, \texttt{w}_2\}$ (see Figures 4.7 and 4.8). Therefore, their characteristic predicates can be gathered: the corresponding departure predicate is $(\texttt{w}_1$ `& full stop) | (`$\texttt{w}_2$ `& full stop)` $\Leftrightarrow$ `d=1 & full stop`. This means that from any state satisfying `d=1 & full stop`, none of the states in $\texttt{s}_1$, $\texttt{s}_2$ or $\texttt{w}_2$ can be reached.

We have now gathered the predicates of the couples (state, intervention(s)) that have the same permissiveness impact. We can now gather the predicates of the states that are not reachable from the same departure predicate (see Algorithm 4). We call it the arrival predicate.

From Algorithm 3, we know the departure predicate of all the states that are not universally reachable (i.e., s is such as there exists $(s_j, i_j), \mathcal{PI}(s_j, i_j) = s$). We then

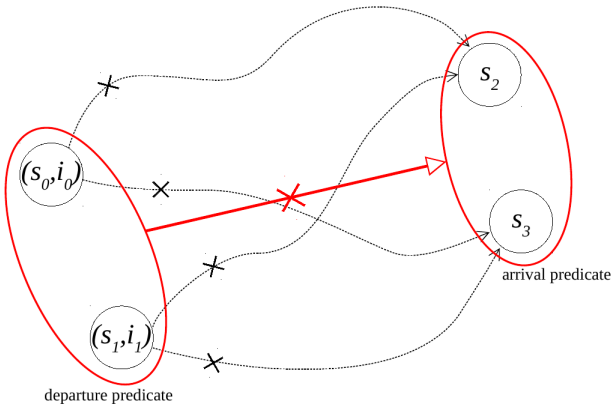

Figure 4.9: Simplification of the departure and arrival predicates

Algorithm 3: Simplifying the departure predicates for same $\mathcal{PI}$

Input: $\mathcal{PI}$, R, $\mathcal{A}_R = (S, T_R, s_0)$

j=0, departure_pred_list=[], arrival_pred_list=[], couple_checked=[]
for $(s_k, i_{s_k}) \in (S_w, \mathcal{P}(R(S_w)))$ *and* $(s_k, i_{s_k}) \notin$ *couple_checked* **do**
  arrival_pred_list[j]=$\mathcal{PI}(s_k, i_{s_k})$
  departure_pred_list[j] = $s_k \wedge i_{s_k}$
  for $(s_l, i_{s_l}) \in (S_w, R(S_w)) \wedge (s_l, i_{s_l}) \notin$ *couple_checked* **do**
    if $PI(s_l, i_{s_l}) = PI(s_k, i_{s_k})$ **then**
      departure_pred_list[j] = departure_pred_list[j] $\vee s_l \wedge i_{s_l}$
      couple_checked+=(s_l, i_{s_l})
    end
  end
  couple_checked+=(s_k, i_{s_k})
  j++
end
departure_pred_list=Simplify(departure_pred_list)

Output: departure_pred_list, arrival_pred_list

compare the departure predicates (Algo. 4, line 7). If two departure predicates are identical, the we can gather their permissiveness impact, i.e, the characteristic predicates of the states that are not reachable from them (Algo. 4, line 8)

In Figure 4.9, the states $\mathtt{s}_2$ and $\mathtt{s}_3$ are both non-reachable from the set of states with the departure predicate represented in red. We gather the characteristic predicated of $\mathtt{s}_2$ and $\mathtt{s}_3$ to compute the arrival predicate. This way we will obtain the information of the non-existing paths between sets of states, one set satisfying the departure predicate, and the other the arrival predicate.

Algorithm 4: Simplifying the arrival predicates for the same departure predicate

Input: $\mathcal{PI}$, R, $\mathcal{A}_R = (S, T_R, s_0)$, departure_pred_list, arrival_pred_list

```
1  j=0, departure_pred_list_simple=[], arrival_pred_list_simple=[], index_checked=[]
2  for k=0..length[departure_pred_list]-1, k ∉ index_checked do
3      departure_pred_list_simplif[j]=departure_pred_list[k]
4      arrival_pred_list_simplif[j] = arrival_pred_list[k]
5      index_checked+=k
6      for l=k..length[departure_pred_list]-1 do
7          if departure_pred_list[k]==departure_pred_list[l] then
8              arrival_pred_list_simplif[j] = arrival_pred_list_simplif[j] ∨ arrival_pred_list[l]
9              index_checked+=l
10         end
11     end
12     j++
13 end
14 arrival_pred_list_simplif=Simplify(arrival_pred_list_simple)
```

Result: arrival_pred_list_simplif, departure_pred_list_simplif

For the example of SI_3, after simplification, we obtain the following result: there is no path from states satisfying the departure predicate `d=1 & full stop` to the states satisfying the arrival predicate `d=2 | d=1 & v=1`. This is represented in Figure 4.10. We can see that there is no path from the state $\mathtt{w}_2$ to itself. The intervention full stop in $\mathtt{w}_2$ forces the velocity to zero.

The simplification tool returns, for the invariant SI_3:

```
-- Non-existing paths:
-- No path from d = 1 & full_stop ;
-- To d = 2 | d = 1 & v = 1;
```

4.3.3 Implementation

The diagnosis algorithms presented in Section 4.3.1, as well as the simplification algorithms presented in Section 4.3.2 have been implemented in Python3.6 (with a gateway to z3 version 4.6.2). It is available online at [SMOF, 2018]. For the diagnosis, the tools takes as input the model of the invariant along with the strategy to analyze. It returns

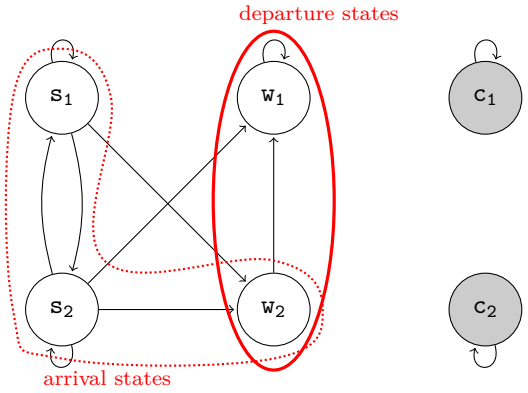

Figure 4.10: Non-existing paths for the invariant SI_3

three files. One contains the list of the simple permissiveness that are violated. One contains the list of the universal permissiveness that are violated and from what simplified departure set. The third one contains the list on the non-existing paths from the departure sets (simplified) to the arrival sets (simplified).

The gateway to z3 contains a parser-rewriter that reads the diagnosis results, write the in SMT language and, after the simplification is done, rewrite them in SMV language.

4.4 Application to Examples

In this section we illustrate the diagnosis of the permissiveness deficiencies of a strategy on two examples extracted from the case study presented in Chapter 2

4.4.1 SI_I: The Arm Must Not Be Extended When The Platform Moves At A Speed Higher Than $speed_{max}$.

Two observations are used for this invariant:

- the speed of the platform, `s` $\in\{[0, s_{max} - margin[$, $[s_{max} - margin, s_{max}[$, $[s_{max}, \infty[\}$=`{0,1,2}`;
- the position of the arm, `a` $\in \{folded, unfolded\}$=`{0,1}`;

Only one intervention is available, triggering the platform brakes. This intervention prevents the speed from increasing.

For this invariant, a single strategy R satisfying the safety property only can be synthesized. This strategy is represented in Figure 4.11.

The diagnosis is performed on this strategy, and the result is represented in Figure 4.12. The state `w2: s = 2 & a = 0` (circled in blue in Figure 4.12) is not reachable: $\mathcal{PD}_{simple}(R) = \mathcal{PD}_{univ}(R) =$`w2`. The permissiveness impacts are: $\mathcal{PI}$`(w1, brakes)=`$\mathcal{PI}$`(w3, brakes)` (circled in dotted blue in Figure 4.12). After the simplification, the tool returns:

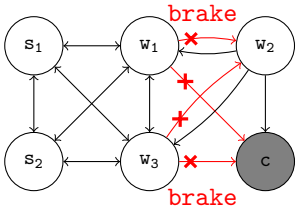

Figure 4.11: Safe and non-permissive strategy for the invariant SI_I (repeated from Figure 2.13)

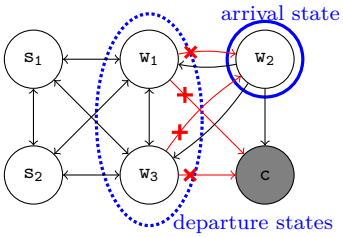

Figure 4.12: Diagnosis for the non-permissive strategy for SI_I

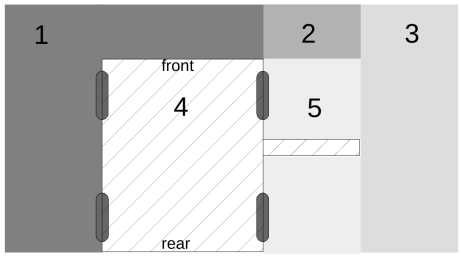

Figure 4.13: Obstacle occupation zones around the robot (repeated from Figure 2.9)

```
-- Non-existing paths:
-- No path from s = 1 & brake;
-- To s = 2 & a = 0;
```

The simplification of the results gives the conclusion that there is no path from the set of states satisfying `s=1 & brake` (i.e., w_1 and w_3) to the state satisfying `s=2 & a=0` (i.e., w_2). Indeed, when the brakes are triggered when `s=1` ($speed_{max} - margin \leq s < speed_{max}$), the speed cannot increase. The state w_2: `s=2 & a=0`, in which $s \geq speed_{max}$ with the arm folded therefore cannot be reached. However, this state is useless for the system to perform its tasks, the strategy could be acceptable. We will see in Chapter 5 how to tune the permissiveness requirements to only require the reachability of states that are necessary for the system's functionalities.

4.4.2 SI_4: The Robot Must Not Collide With An Obstacle.

For this invariant, a satisfying strategy was presented in Chapter 2. However, we explore here the results that would be found if only the braking intervention was available. Only the braking intervention will not be sufficient to ensure the permissiveness as no lamp would be allowed to pass under the arm (i.e., get closer than the braking distance). Let us see if the diagnosis confirms this intuition. As a reminder, the partition in zones of the environment of the robot is represented in Figure 4.13.

The non-permissive strategy (brake whenever an obstacle is nearby) that we propose to analyze is the following:

```
DEFINE flag_brake:= z1=1 | z2!=0 | z3=1 | z4=1 | z5!=0;
```

The result of the diagnosis for the simple permissiveness gives the following results:

```
-- Simple permissiveness deficiencies:
DEFINE simple_perm_def := z4 = 0 & z5 = 2 & v = 1;
```

The states satisfying `z4 = 0 & z5 = 2 & v = 1` correspond to states where the zone 4 (the robot itself) is empty, a low obstacle (a lamp) is in the zone 5 (under the arm), and the robot is moving. If this state cannot be reached, the robot will never be able to measure the lights: it needs to move along lamps (in zone 5) to be able to measure the line of lamps. This permissiveness deficiency is not acceptable.

The results found for the universal permissiveness tell us that 89 states are not universally reachable. After simplification, the non-existing paths are given as follows:

```
--Non-existing paths:
--No path from z2 = 2 & v = 0 & brake | z5 = 2 & v = 0 & brake;
--To z2 = 0 & z5 = 0 & v = 0 | z4 = 0 & z5 = 0 & v = 1;
-----------------------------------------------------------------------
--No path from z5 = 2 & v = 0 & brake;
--To z2 = 1 & z5 = 0 & v = 0 | z2 = 2 & z5 = 0 & v = 0;
-----------------------------------------------------------------------
--No path from z2 = 2 & v = 0 & brake;
--To z2 = 0 & z5 = 1 & v = 0 | z2 = 0 & z5 = 2 & v = 0;
-----------------------------------------------------------------------
--No path from z2 = 0 & z5 = 2 & v = 0 & brake;
--To z2 = 2 & z5 = 1 & v = 0;
-----------------------------------------------------------------------
--No path from z2 = 2 & z5 = 0 & v = 0 & brake;
--To z2 = 1 & z5 = 2 & v = 0;
-----------------------------------------------------------------------
--No path from z2 = 2 & z5 = 0 & v = 0 & brake | z2 = 0 & z5 = 2 & v = 0 &
brake;
--To z2 = 2 & z5 = 2 & v = 0;
```

Let us consider for example the first result:
There is no path from:
`z2=2 & v=0 & brake | z5=2 & v=0 & brake`
to
`z2=0 & z5=0 & v=0 | z4=0 & z5=0 & v=1.`
The departure predicate means that the robot is stopped, the brakes triggered, and there is a low obstacle in the zone 2 or 5, i.e., on the side of the arm. From these states, there is no path to the states where the zones 2 and 5 are empty and the robot is stopped, or to the states where the zone 4 (the robot) and 5 are empty while the robot is moving.

What we understand is that as soon as a low obstacle (a lamp) is in the zone 2 or 5, it is not possible to empty these zones: the brakes are triggered, stopping the robot and as the lamps are not moving, the brakes will stay triggered and the robot stopped.

The others results confirm this. For every one, we see that when a lamp is under the robot's arm, the brakes are triggered, it is not possible to empty or fill the zones 2 and 5 (i.e., to move along the lamps). To perform the measuring task, the robot needs to move along the lamps so that the sensor passes above them. The permissiveness deficiencies are not acceptable in this case.

This suggests that defining new interventions could be interesting in this case. This is indeed what we did when manually analyzing this invariant in Chapter 2. However, we needed several modeling iterations to solve the case where the diagnosis tool would have avoided that and helped us design the appropriate interventions more easily. In the Chapter 6, we explore a way of automatically suggest appropriate interventions for the synthesis.

4.5 Conclusion

In this chapter, we presented and formalized the two concepts of permissiveness deficiencies of a strategy and permissiveness impact of a couple (state, intervention(s)). We detailed the algorithm that we use to evaluate these two parameters. We also detailed a set of simplifications that we apply to the results with the help of the solver z3, that very much increase their readability. We saw that this allows us to identify why the synthesis fails to return a safe and permissive strategy. This diagnosis algorithm has been implemented and successfully applied on examples (two of them presented in this chapter).

The diagnosis can help the user to choose how to solve this issue. In the first example example, we saw that the permissiveness deficiencies of the analyzed strategy are acceptable. The solution can be to tune the permissiveness requirements to adapt them to the functionalities. This will be explored in Chapter 5. In the second example, the permissiveness deficiencies are not acceptable. Then, a solution is to change the available interventions. In Chapter 6, we propose a method to define new interventions for the synthesis.

Take Aways:

- The permissiveness deficiencies qualify the permissiveness loss imposed by a strategy;
- The permissiveness impact of a couple (state, intervention) can help the user identify which intervention, when it is triggered, is responsible for a permissiveness loss;
- The gateway to the simplification facility of z3 increases the readability.

Chapter 5

Tuning the Permissiveness

Contents

When SMOF fails to find a permissive strategy, one solution is to change the permissiveness requirements. The first objective of this chapter is to propose a way to adapt the permissiveness to the system's expected functionalities. The new permissiveness properties can then be used for the synthesis: the reachability of a subset of states is required, thereby giving more flexibility to the synthesis. However, even with these new permissiveness properties, it happens that no strategy can be synthesized. The second objective of this chapter is to propose a way to weaken the permissiveness requirements with a traceable impact on the system's ability to performs its tasks.

In this chapter, we present a template for the definition of custom permissiveness, that allows the user to adapt the permissiveness to the system's functionalities, without having to use CTL (Section 5.1). We also present three ways to weaken the permissiveness and explain what the weakening implies for the system's ability to function (Section 5.2). We apply the definition and weakening of custom permissiveness on three examples, and analyze the results (Section 5.3). We conclude in Section 5.4.

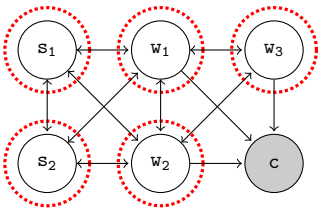

Figure 5.1: Required reachability for generic permissiveness

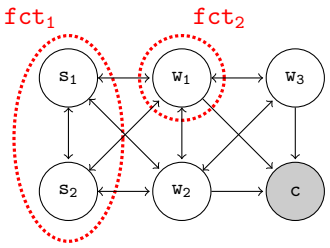

Figure 5.2: Required reachability for custom permissiveness

5.1 Defining Custom Permissiveness Properties

The generic definition of the permissiveness properties provided by SMOF requires the reachability of *all* the non-catastrophic states. In this section, we explain how to change this definition to only require the reachability of a subset of states, adapted to the system's functionalities, therefore giving more flexibility to the synthesis.

5.1.1 From Generic to Custom Permissiveness

By default, SMOF requires the universal permissiveness, i.e., all the non-catastrophic states must remain reachable from all the other non-catastrophic states. This is represented with the red circles on the abstract example in the Figure 5.1. As a result, the synthesis algorithm prunes any strategy that would cut all paths to a non-catastrophic state.

However, all the non-catastrophic states may not be necessary to reach for the accomplishment of the functions of the system. To give an example, let us consider the invariant SI_1, stating that the system velocity should never reach a maximal absolute value v_{max}. The synthesis would reject any strategy where the paths are cut to warning states with values close to v_{max} (i.e., within the margin $v_{max} - margin$). But the cruise velocity of the system, used to accomplish its mobility functions, is typically much lower than v_{max} and $v_{max} - margin$. Requiring the universal reachability of the warning states is useless in this case, since getting close to v_{max} is not a nominal behavior. The system operation could well accommodate a safety strategy that forbids evolution to close-to-catastrophic velocity values.

From what precedes, it may seem that we could simply modify the generic definition of permissiveness to require universal reachability of safe states only, excluding warning states. However, this would not work for all systems, as demonstrated by the collision invariant SI_4 presented in Chapter 2. For this invariant, some warning states *do* correspond to a nominal behavior and are essential to the accomplishment of the maintenance mission. More precisely, the robot is intended to control the intensity of lights along the airport runways. The light measurement task is done by moving very close to the lights, which, from the perspective of the anti-collision invariant, corresponds to a warning

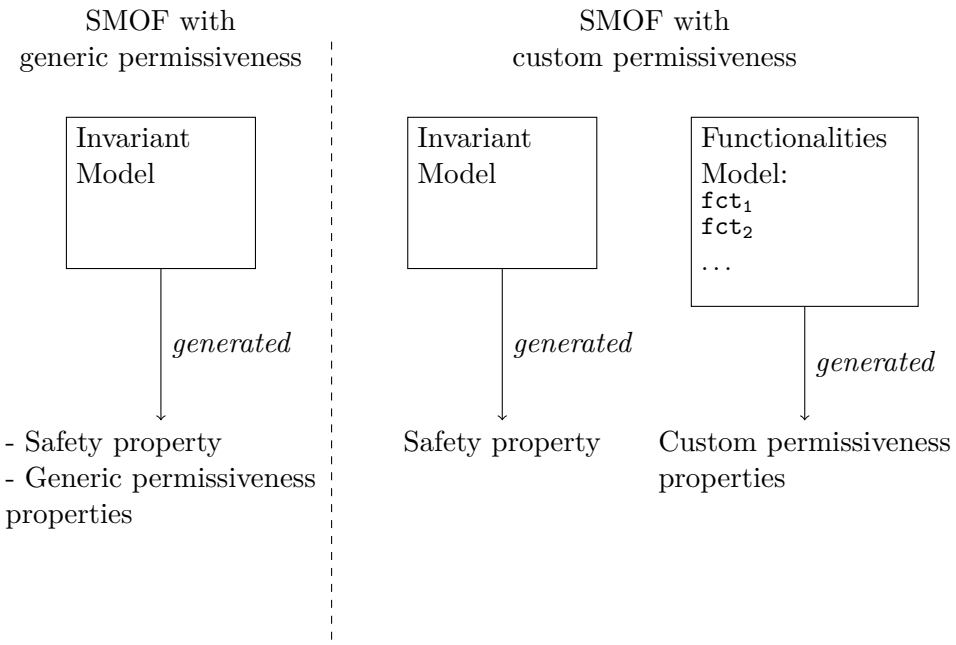

Figure 5.3: Invariant and functionalities models

state. Any safety strategy removing reachability of a close-to-catastrophic distance to the lights would defeat the very purpose of the robot.

Actually, there is no *generic* definition of permissiveness that would provide the best trade-off with respect to the system functions. We would need to incorporate some application-specific information to tune the permissiveness requirements to the needs of the system, i.e., create some *custom* permissiveness properties. In the abstract example of Figure 5.2, the custom permissiveness requires the reachability of the sets of states in red, corresponding to two functionalities $\texttt{fct}_1$ and $\texttt{fct}_2$. The reachability of the states $\texttt{w}_2$ and $\texttt{w}_3$ is not required any more. A strategy discarding all paths to these states is satisfying with regard to the custom permissiveness.

We propose a way to introduce such custom permissiveness properties into SMOF, allowing more strategies to be found and facilitating the elicitation of trade-offs in cases where some functionalities must be restricted due to safety concerns. The custom permissiveness properties are associated to the functionalities the system is supposed to perform. These functionalities need to be formalized. From this formalization, we propose to extract CTL properties —to be used in the synthesis— following a simple template.

5.1.2 A Formal Model for the Functionalities

In SMOF, each invariant is modeled independently. The generic permissiveness properties (as well as the safety property) are automatically generated from the model of the invariant (see Figure 5.3).

Now, we want the custom permissiveness properties to correspond to the system's expected functionalities. While generic permissiveness properties apply to all non-catastrophic states, we choose to express the custom ones as the reachability of a subset of states, the ones that are essential to the system's functionalities. We introduce a state model dedicated to the functionalities, giving a view of the essential/non-essential states. This model is different from the invariant model (see Figure 5.3), that gives a view of the safe/warning/catastrophic states.

The state model for the functionalities is defined as a set of variables $\mathcal{V}_f$ partitioned in classes of values of interest. For instance, let us consider the functionality f, which requires a continuous observable variable v (e.g., velocity, or position of a tool) to reach a given value v_{req} (e.g., cruise velocity, vertical position) with some tolerance δ. The domain of the variable v would be partitioned into three classes: `0` corresponding to values lower than $V_{req} - \delta$, `1` to values in $[V_{req} - \delta, V_{req} + \delta]$, `2` to values greater than $V_{req} + \delta$. We can create $\texttt{v}_{\texttt{fct}} \in \{\texttt{0,1,2}\}$, the abstract continuous variable encoding the partition from the point of view of the functionality. It will be encoded using the predefined `continuity` module offered by SMOF (see Chapter 2). Generally speaking, the modeling can reuse the syntactic facilities offered for the safety model, also defined in terms of observable variables, classes of values of interest and evolution constraints.

Note that the partition drawn for the variable v used in the functionalities model may be different from the one defined for the variable v in an invariant. Typically for the velocity variable, we would have, for a movement functionality at a required cruise speed v_{cruise}, $\texttt{v}_{\texttt{fct}} \in \{[0, v_{cruise} - \delta[, [v_{cruise} - \delta, v_{cruise} + \delta[, [v_{cruise} + \delta, \infty[\} = \{\texttt{0,1,2}\}$, and for the overspeed invariant $\texttt{v}_{\texttt{inv}} \in \{[0, v_{max} - margin[, [v_{max} - margin, v_{max}[, [v_{max}, \infty[\} = \{\texttt{0,1,2}\}$.

The purpose of the functionality model is to allow the user to conveniently specify sets of states that are essential to a given functionality. A set of states is introduced by a predicate req over a variable or a set of variables in $\mathcal{V}_f$. In the velocity example above, the user would specify that the functionality requires $\texttt{v}_{\texttt{fct}}\texttt{=1}$. Each functionality may introduce several requirements, i.e., several essential sets of states. For instance, a "move" functionality could have two separate requirements, one for cruise motion and one for slow motion.

The list of required predicates can then be extracted to produce permissiveness properties of the form: $AG(EF(req))$. We choose to reuse the same template as the one for the universal permissiveness. However, we now require the reachability of sets of states, rather than the reachability of every individual state. For example, we have no reachability requirement on states satisfying $\texttt{v}_{\texttt{fct}}$ `= 2`, and may accommodate strategies discarding some of the states $\texttt{v}_{\texttt{fct}}$ `= 1` provided that at least one of them remains reachable. This list of properties replaces the generic ones for the synthesis: $Perm = \{AG(EF(req_1)), AG(EF(req_2)), \dots\}$.

5.1.3 Binding Together Invariants and Permissiveness

The permissiveness and the safety properties are defined using two different state models. Some of the abstract variables used in those state models represent the same physical observation, or dependent ones (for example the velocity for a movement functional-

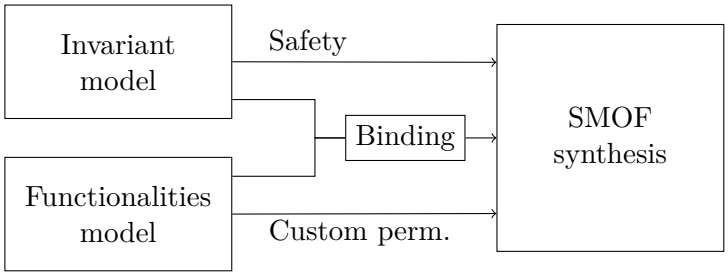

Figure 5.4: Binding of the functionalities model and the invariant model

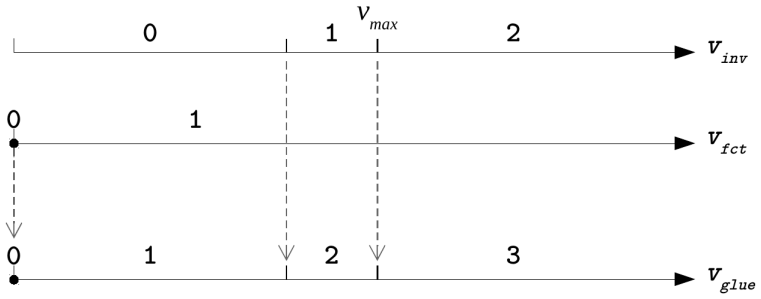

Figure 5.5: Binding for two velocity variable

ity and an overspeed invariant). To connect the invariants and functionalities models, thereby ensuring a consistent evolution of the variables, we have to bind their variables. The invariant model, the functionalities model and the binding are then injected into SMOF's synthesis (see Figure 5.4).

Two types of bindings can be used: physical dependencies (e.g., speed and acceleration), or the use of the same observation with two different partitions.

In the first case, we specify the constraints on transitions (using the NuSMV keyword `TRANS`) or on states (`INVAR`). For example, for observations of speed and acceleration, we would write `TRANS next(acc)=0` $\rightarrow$ `next(speed)=speed`, i.e., if the acceleration is null, the speed cannot change.

In the second case, we need to introduce a "glue" variable to bind the different partitions. This variable will be partitioned in as many intervals as needed. The different intervals will be bound with a specification on the states. For example, let us assume we have an invariant and a functionality using a velocity variable, and the partition used for the invariant is $\mathtt{v_{inv}}$=`{0,1,2}` where `0`: stationary or slow, `1`: medium and `2`: high (above a maximum velocity value v_{max}), and the one used for the functionality is $\mathtt{v_{fct}}$=`{0,1}` where `0`: stopped and `1`: moving. We introduce a continuous "glue" variable partitioned as $\mathtt{v_{glue}}$=`{0,1,2,3}` (see Figure 5.5). The binding through the glue variable is specified as follows:

```
--Continuity (value_min, value_max, value_init)
DEFINE v_glue:  Continuity(0,3,0);
INVAR v_glue=0 ↔ v_inv=0 & v_fct=0;
INVAR v_glue=1 ↔ v_inv=0 & v_fct=1;
INVAR v_glue=2 ↔ v_inv=1 & v_fct=1;
INVAR v_glue=3 ↔ v_inv=2 & v_fct=1;
```

Note that those two binding approaches, by adding constraints or glue variables, are also used in the standard SMOF process when merging models of different safety invariants (the consistency analysis step, presented in Chapter 2).

5.2 Restricting Functionalities

When no strategy can be synthesized despite the use of the custom permissiveness properties, the user may have to accept restrictions of functionalities. A first step is to use the diagnosis tool, in order to identify which functionality was blocking the synthesis, i.e., which functionality to restrict. Then, we propose three ways of weakening the custom permissiveness properties resulting in different levels of restrictions of the functionalities.

5.2.1 Diagnosing the Permissiveness Deficiencies with the Custom Properties

When no safe and permissive strategy exists with custom permissiveness properties, the diagnosis tool can be used similarly as presented for the generic permissiveness.

We re-use the Algorithms 1 and 2 presented in Chapter 4, but we adapt them to calculate the permissiveness deficiencies and impact for the functionalities: instead of checking the reachability of all the states, we only check for each functionality the reachability of at least one state satisfying the corresponding required predicate.

The permissiveness deficiencies of a strategy, with the custom permissiveness, is the list of the functionalities predicates that are not reachable, i.e., no state satisfying the predicate is reachable. It also encompasses the simple and universal reachability. $\mathcal{PD}_{simple}$ is the list of functionalities predicates that cannot be satisfied from the initial state. $\mathcal{PD}_{univ}$ is the list of functionalities predicates that cannot be satisfied from some non-catastrophic state(s).

The permissiveness impact here points at the couple (state, intervention) from which a functionality predicate is not reachable. For a non-satisfied functionality f (req its characteristic predicate), $s \in S_w$ (the set of warning states), $i \in \mathcal{P}(R(s))$ (R the set of interventions applied in the state s), $\mathcal{PI}(s,i) = req$ if there is no path from the state s with the intervention i active to a state satisfying the predicate req. A step of simplification if performed as well, resulting in a list of non-existing paths between some sets of states to the states satisfying the functionalities predicates.

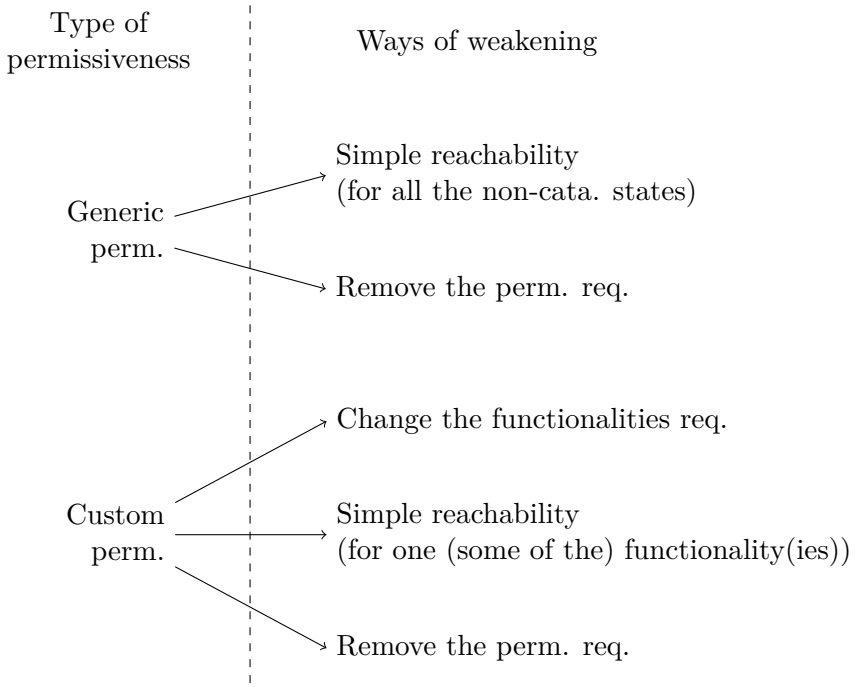

Figure 5.6: Weakening generic and custom permissiveness

5.2.2 Weakening the Permissiveness Property

Custom permissiveness is weaker than SMOF's generic permissiveness, since we get rid of non-essential reachability requirements. As a result, the strategy synthesis tool may return relevant strategies that would have been discarded with the generic version.

Still, it is possible that the custom requirements do not suffice, and that no strategy is found by the tool. We want to make it possible that the user restricts (some of) the functionalities, i.e., further weakens permissiveness. This may change the system's objectives, or increase the time or effort it takes to achieve the objectives. Functionalities can be restricted in several ways. We consider three of them here. The first one is to redefine the required functionalities. The second and third ones are inspired from the possibilities provided by SMOF for the generic permissiveness (see Figure 5.6).

Change the functionalities requirement: One of the req predicates can be weakened to correspond to a larger set of system states. The permissiveness property becomes req', with $req \implies req'$. For example, the "move" functionality can be restricted to "move slowly", where req' means that v only needs to reach a velocity $v_{req'}$ lower than the initially specified cruise one. The partition drawn for the modeling of req may not remain valid. The functionality model, as well as the binding with the safety model have to be changed.

Applying this restriction requires a complete remodeling of the affected functionality. A knowledge of the physical values used for the partition is needed, to identify the

ordering between the classes of values of the safety and the functionality model while doing the binding.

Use the simple reachability: It is possible to replace the universal permissiveness property by the simple one, $EF(req)$. This weak form of permissiveness is already offered by the generic version of SMOF, but it applies to all individual states. Weakening the custom permissiveness properties associated to a functionality —from universal to the simple permissiveness— means that the monitor's intervention may be irreversible with regard to this functionality. With the custom list of *req* predicates, the user can decide for which of these predicates simple permissiveness would be acceptable. For example, the "move" functionality could become impossible after an emergency stop has been triggered, but other functionalities like manipulating objects would have to be preserved.

Removing a functionality from the requirements: The user can also simply remove the functionality from the requirements. For example a "manipulation while moving" functionality is no longer required. Here, the corresponding specified requirement is simply deleted, and the synthesis run again without it. This ultimate restriction step can show the user that the intended functionality is not compatible with the required level of safety. This information can be propagated back to the hazard analysis step and used to revise the design of the system or its operation rules. Again, not all of the requirements may need a restriction. The functionalities that are not restricted are guaranteed to remain fully available despite the monitor's intervention.

5.2.3 Automatic Generation and Restriction of Permissiveness Properties

Integrating the management of custom permissiveness into SMOF was possible without any major change of the core toolset. Only the front-end had to be updated.

The user does not have to write CTL properties but just the predicate characterizing the essential states to reach. Concretely, the functionalities requirements are specified following the template in Figure 5.7.

For the tool to generate the CTL properties following the right template (universal permissiveness, simple permissiveness or removed), a restriction level is attributed to all the functionalities: 0 when the functionality is not restricted, 1 when the simple permissiveness is used and 2 when the functionality is removed. A configuration file for the SMOF synthesis is added, containing the following information:

```
- type of permissiveness used (generic:  0 or custom:  1)
- if custom permissiveness, number of functionalities defined
- list of the functionalities & level of restriction
```

For instance, for a functionality model with a "move" functionality (with a restriction level of 1) and a "stop" functionality (with a restriction level of 0, i.e., non-restricted), the configuration file is as follows:

```
--Definition of variables (using the same facilities as in the invariant model)
VAR name_var :  type_var;
--Definition of the dependency constraints between variables (same as invariant
model)
TRANS ...;
INVAR ...;
--Definition of the functionalities requirements
DEFINE name_fct := predicate_fct;

--Define the glue variables
VAR ...;
--Define the gluing constraints
TRANS ...;
INVAR ...;
```

Figure 5.7: Template for the functionalities model

```
1
2
move 1
stop 0
```

Auto-completion facilities (implemented in C++, for integration purposes with SMOF) allow the generation of the CTL properties from the functionalities model and the configuration file. The user can vary the level of restriction of the functionalities by changing the associated number in the configuration file. Once written, the functionality model does not have to be modified to take into account restrictions.

The core modeling approach and strategy synthesis algorithm remain unchanged.

5.3 Application on Examples

In this section we apply the definition of custom permissiveness properties for two examples that we studied in Chapter 2. The models for the mentioned examples can be found online [SMOF, 2018].

5.3.1 SI_I: The Arm Must Not Be Extended When The Platform Moves At A Speed Higher Than $speed_{max}$.

Modeling: We consider the invariant SI_I: the arm must not be extended when the speed is higher than s_{max}. The available observations are $\mathtt{s_{inv}}$, the speed of the platform; and $\mathtt{a_{inv}}$, the position of the arm. Note that the variables names are extended with the index `inv` to specify that they are the variables used for the safety invariant model. As a reminder, the detail of the partition for these variables is shown in Table 5.1, and the catastrophic state can be expressed as `cata:` $\mathtt{s_{inv}}$`=2 &` $\mathtt{a_{inv}}$`=1` (high speed with extended arm).

Speed of the platform	**Real speed interval**	**Discrete variable**
Low	$s_{inv} < s_{max} - m$	$\mathtt{s_{inv}} = 0$
Within the margin	$s_{max} - m \leq s_{inv} < s_{max}$	$\mathtt{s_{inv}} = 1$
Higher than the maximum allowed value	$s_{inv} \geq s_{max}$	$\mathtt{s_{inv}} = 2$
Position of the arm		**Discrete variable**
Not extended beyond the platform		$\mathtt{a_{inv}}=0$
Extended beyond the platform		$\mathtt{a_{inv}}=1$

Table 5.1: Partitioning of the variables $\mathtt{s_{inv}}$ and $\mathtt{a_{inv}}$ (repeated from Table 2.6)

Speed of the platform	**Real speed interval**	**Discrete variable**
Null or lower than the minimum cruise speed	$s_{fct} < s_{cruise\_min}$	$\mathtt{s_{fct}} = 0$
At the minimum cruise speed or higher	$s_{fct} \geq s_{cruise\_min}$	$\mathtt{s_{fct}} = 1$
Position of the arm		**Discrete variable**
Folded		$\mathtt{a_{fct}}=0$
Extended beyond the platform		$\mathtt{a_{fct}}=1$

Table 5.2: Partitioning of the variables $\mathtt{s_{fct}}$ and $\mathtt{a_{fct}}$

To express the relevant permissiveness properties, we identify in the specifications what functionalities are related to the invariant. Let us consider the variables involved in $\mathrm{SI_I}$. The $\mathtt{s_{inv}}$ variable is an observation of the speed of the mobile platform, in absolute value. The system is supposed to move around the workplace to carry objects, i.e., the speed must be allowed to reach a minimal cruise speed value s_{cruise_min}, from any state. To model this functionality we introduce the $\mathtt{s_{fct}}$ variable, which will be partitioned as showed in Table 5.2. Note that the variables names are extended with the index `fct` to specify that they are the variables used for the functionalities model. This functionality can be expressed as `cruise motion:` $\mathtt{s_{fct}}=1$, $\mathtt{AG(EF(s_{fct}=1))}$ being the associated permissiveness property automatically generated by SMOF. The system must also be able to stop or move slowly, thus another functionality, `slow_motion`, is expressed: $\mathtt{s_{fct}}=0$, corresponding to $\mathtt{AG(EF(s_{fct}=0))}$. Also, the $\mathtt{a_{inv}}$ variable models whether the manipulator arm is extended beyond the platform or not. To handle objects, the arm must be allowed from any state to reach a state where the arm is extended beyond the platform, and a state where the arm is folded. We introduce the variable $\mathtt{a_{fct}}$ which is partitioned as showed in Table 5.2. The two functionalities related to the arm are the `arm_extension:` $\mathtt{a_{fct}}=1$ and the `arm_folding:` $\mathtt{a_{fct}}=0$. The SMOF model for these two functionalities is as follows:

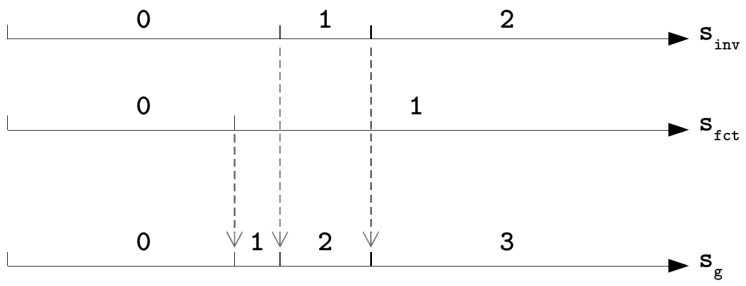

Figure 5.8: Partitioning of the s_g variable

```
--Declaration of the variables
VAR s_fct:  Continuity(0,1,0);
VAR a_fct:  Continuity(0,1,0);
--Definition of the functionalities
DEFINE slow_motion := s_fct=0;
DEFINE cruise_motion := s_fct=1;
DEFINE arm_folding := a_fct=0;
DEFINE arm_extension := a_fct=1;
```

The speed value and arm position are observed in both the invariant model (s_{inv} and a_{inv}) and the functionalities model (s_{fct} and a_{fct}). We need to make their evolution consistent. To do so, we introduce glue variables, s_g and a_g.

For the speed, we have two different partitions as shown in Figure 5.8, one for s_{inv} (with discrete values {0,1,2}) and one for s_{fct} (with discrete values {0,1}). The resulting glue variable s_g will then have four values. We thus have the formal definition:

```
--Continuity(value_min, value_max, value_init)
VAR s_g:  Continuity(0,3,0);
INVAR s_g=0 ↔ s_inv =0 & s_fct=0;
INVAR s_g=1 ↔ s_inv =0 & s_fct=1;
INVAR s_g=2 ↔ s_inv =1 & s_fct=1;
INVAR s_g=3 ↔ s_inv =2 & s_fct=1;
```

For the arm variable, it is much simpler:

```
VAR a_g:  Continuity(0,1,0);
INVAR a_g=0 ↔ a_inv=0 & a_fct=0;
INVAR a_g=1 ↔ a_inv=1 & a_fct=1;
```

The two available interventions are to brake, which affects the speed, and to block the extension of the arm. These interventions have been detailed in Chapter 2, but as a reminder, their definitions are as follows:

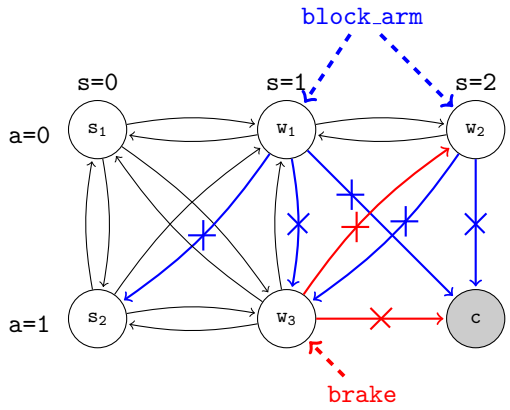

Figure 5.9: Single strategy synthesized for the invariant SI_I with generic permissiveness properties (repeated from Figure 2.14)

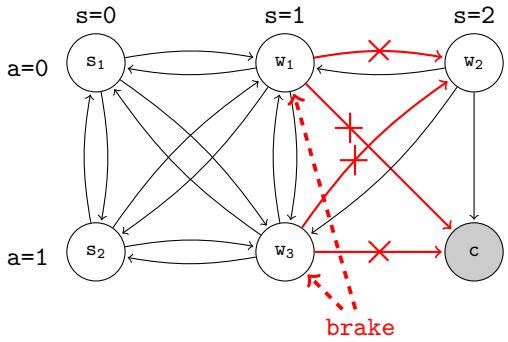

Figure 5.10: Additional strategy synthesized for the invariant SI_I with the custom permissiveness properties.

```
--Interv(stat_precond, seq_precond, effect)
VAR brake:  Interv(TRUE, s=0, next(s)!=s+1);
VAR block_arm:  Interv(a=0, TRUE, next(a)=0);
```

Results: Let us compare the results obtained without and with the approach through the definition of custom permissiveness. In the first case, we use the generic permissiveness, i.e., the reachability of every non-catastrophic state (the states {s1, s2, w1, w2, w3} in Figure 5.9), from every other non-catastrophic state. Only the strategy presented in Chapter 2 is both safe and permissive (see Figure 5.9).

In the second case, we replace the generic permissiveness with the use of the custom permissiveness properties `cruise motion`, `slow motion`, `arm folding` and `arm extension` specified before. They amount to only require the reachability of the states {s1, s2}. After running the synthesis, in addition to the previous strategy we have a strategy only using the braking intervention (see Fig. 5.10). This can be preferable in some cases, as the use of the arm is then never impacted and even if the monitor triggers the brakes the system can keep manipulating objects. This strategy couldn't be found with the generic permissiveness as it removes the reachability of w2, a useless state from the perspective of the intended functionalities.

5.3.2 SI_{II}: A Gripped Box Must Not Be Tilted More Than α_0.

We explore here a different formalization than the one presented in Chapter 2 (the complete formalization can be found in [SMOF, 2018]). Three observations are used: the angle of rotation of the arm ($\alpha \in \{[0, \alpha_0 - m_1[,\ [\alpha_0 - m_1, \alpha_0[,\ [\alpha_0, \infty[\}$=`{0,1,2}`), the presence of a box (`box` $\in$ `{0,1}`) and the distance between the gripper's fingers ($d_{gripp} \in \{[0, d_{max} - m_2[, [d_{max} - m_2, d_{max}[,\ [d_{max}, \infty[\}$=`{0,1,2}`). The distance between the

gripper's fingers was not used in the model presented in Chapter 2. The catastrophic state is that a box is held with the arm tilted, i.e., `cata: α=2 & box=1`.

Two interventions are available. The first one brakes the arm, i.e., prevents the angle of rotation of the arm from increasing. The second one locks the gripper, i.e., prevent it from closing.

The automated synthesis with the generic permissiveness finds a safe strategy but fails to return a safe and permissive strategy. We now revisit this model by specifying custom permissiveness properties for a manipulation functionality (carrying a box at a low rotation angle). Using the custom permissiveness, the tool successfully synthesizes a strategy. It combines the braking of the arm and the lock of the gripper to maintain the invariant while permitting functionality.

5.3.3 SI_3: The Robot Must Not Enter A Prohibited Zone.

Modeling: We consider here the invariant SI_3: the robot must not enter a prohibited zone, that has been detailed in Chapter 2. The observations used are `d`, the distance to the prohibited zone, `d: {0, 1, 2}` and the velocity v_{inv}: `{0, 1}`. The catastrophic state is expressed as `cata: d=0`. The only available intervention here is the full stop intervention, which stops the robot completely.

In this case, for the functionalities we just need to specify that the robot needs to reach a state where it is moving, and a state where it is stopped. We model the custom permissiveness with `move:`v_{fct}`=1` and `stop:` v_{fct}`=0` where v_{fct} represents the robot moving or stopped. This variable is directly bound to the v_{inv} variable with a glue variable v_g as:

```
INVAR v_g=0 ↔ v_inv=0 & v_fct=0;
INVAR v_g=1 ↔ v_inv=1 & v_fct=1;
```

Results: The synthesis of strategies with the full stop intervention and the `move` and `stop` functionalities does not give any result. The only strategy guaranteeing safety, regardless to the permissiveness is a strategy applying the full stop whenever the robot gets too close to the prohibited zone, represented in Figure 5.11. If we analyze this strategy with our diagnosis tool (see Chapter 4), we obtain the following result (for the universal reachability of the functionalities predicates):

```
--Universal permissiveness deficiencies for functionality move:
--No path from d = 1 & full_stop
--To move
```

This means that the functionality `move` is not reachable from the states satisfying `(d = 1 & full_stop)`. Indeed, the system is stopped close to the prohibited zone and cannot ever move again, i.e., the monitor's intervention is irreversible. In term of automaton, we reached a deadlock state (state w_1 in Figure 5.11).

If we want to improve this strategy while guaranteeing safety, we need to either change the interventions or accept a restriction of the functionality as described in Sec-

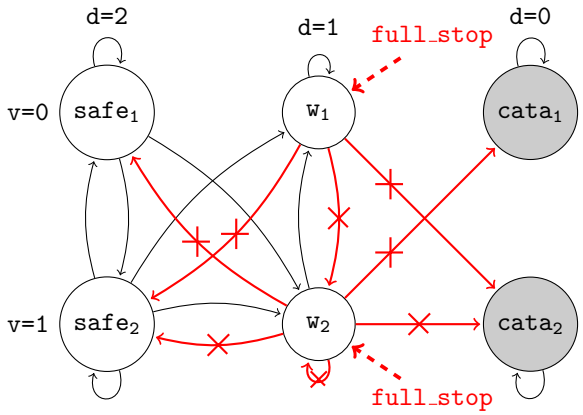

Figure 5.11: Safe strategy synthesized for the invariant SI_3

tion 5.2. Here we consider that we do not have any other intervention, we thus do not have other choice than restricting the functionality.

We choose to weaken the property to the simple permissiveness: we accept the intervention of the monitor to be irreversible, but we still want the functionality to be fully available before the intervention of the monitor (i.e., we change the restriction level to `1` in the configuration file). The property for `stop` remains unchanged. This restriction can seem drastic, but if the controller brings the robot too close to the prohibited zone, we can suspect a malfunction in the control channel. It thus makes sense to stop the robot.

With the restricted permissiveness property, the synthesis that was initially generated is appropriate: it satisfies the safety property and the custom permissiveness properties associated with the functionalities `move` (restricted to simple reachability) and `stop`. Specifying the restriction of functionalities highlights the impact of the monitor intervention on the system ability to function, which was not possible with the use of generic simple permissiveness.

5.4 Conclusion

In this chapter, we overcome an overly stringent definition of the monitor's permissiveness, in proposing a custom definition of permissiveness according to the system's functionalities (the behaviors necessary to fulfill its purposes). The custom permissiveness properties are expressed following a simple template. We require the reachability of a reduced set of states, therefore, more strategies can be synthesized. In the example of the arm extension invariant, the proposed solution provided a new strategy only requiring the use of one intervention instead of two. Also, for the gripper invariant, we could find a solution to a problem which had no solutions with a generic definition of permissiveness properties.

Whenever it is not possible to synthesize a satisfying strategy with regard to permissiveness, we propose an iterative design strategy: we give three ways to adapt functionalities by weakening the permissiveness properties following a template. In these situations, some strategies can often still be found with slight and traceable changes of the functionalities. The impact of the monitor on the robot's operation can thus be qualified and reasoned about.

Integrating the definition and use of custom permissiveness properties with the existing SMOF tooling only required a small change on the front-end. The synthesis algorithm remains unchanged.

The functionalities that would require another type of template are not considered yet, but so far the expressiveness of this template has been sufficient to model the considered functionalities. In further work, it is quite conceivable to integrate other templates for the modeling of more complex functionalities, or of different types of restrictions. Any type of CTL property can be taken into account for the synthesis with minor changes to the tool itself.

Take Aways:

- Definition of custom permissiveness according to system functionalities;
- Restriction of functionality guided by the diagnosis;
- Three possibles ways to restrict functionalities;
- Automatic generation and weakening of properties by the tool;
- It helps for the identification of the trade-off between safety and functionalities due to the monitor.

Chapter 6

Suggestion of Candidate Safety Interventions

Contents

When SMOF indicated that there is no safe and permissive strategy, one solution is to change the available interventions. Defining new interventions can be arduous, as the user needs to know which variable it should impact and how. The objective of this chapter is to define an interactive method for identifying a list of candidate safety interventions to be injected in the synthesis. The interventions are tightly linked to the implementation of the system. Automatically generated interventions may not be implementable, and it may not be possible to use them in a monitor. We want to set the suggestion so that the candidate interventions are as realistic as possible. Also, the user needs to review the suggested interventions and adapt them to the systems possibilities. The suggestion of interventions is a more prospective contribution than the others and is not supported by a tool yet. However the principles are formalized and the core algorithms are defined.

We first discuss how the interventions are defined through their preconditions and effect (Section 6.1). We then present the automatic identification of candidate interventions based on the invariant model, and how the user can review them to adapt them to the system's abilities (Section 6.2). We present the algorithm for the interactive suggestion of interventions in Section 6.3, and apply it manually to an example in Section 6.4. We conclude in Section 6.5.

6.1 Preconditions and Effects of Interventions

In this section we propose to review how the interventions are defined, and to discuss their characteristics.

The interventions are defined through two main parameters: their effect and their preconditions. The preconditions define in what case an intervention can be applied: the history of the system at one step is considered (the sequential precondition), as well as the current state in which the intervention is triggered (the static precondition). They model the physical characteristics that will guarantee that the intervention will have the expected effect. For instance, the intervention "blocking the arm extension", used for the invariant SI_I (the arm must not be extended when the speed is higher than $speed_{max}$) will indeed block the extension only if the arm is not already extended (its static precondition is that the arm is not already extended).

The effect of an intervention models how it modifies the behavior of the system. It impacts the discrete variables, and must respect their dynamics. For instance, the *speed* variable used for SI_I is defined as `speed` $\in \{[0, speed_{max} - m[, [speed_{max} - m, speed_{max}[, [speed_{max}, \infty[\}$=`{0,1,2}`. It is a continuous variable: the value of the discrete variable representing it can only increase by one, decrease by one, or stay the same. It is not possible for an intervention to make the speed value jump from `0` to `2` in one step.

From our experience, most interventions affect only very few (if not only one) variables. Specific changes of the affected variables will be prevented or forced, potentially resulting in the removal of several transitions. For instance, consider the braking intervention specified for the invariant SI_I. The intervention model with SMOF is:

```
-- Interv(static precondition, sequential precondition, effect);
VAR brake:  Interv(TRUE, s=0, next(s)!=s+1);
```

The intervention has no static precondition (`TRUE`). It has one sequential precondition: the margin threshold has just been crossed (`s=0` in the previous state). This models the fact that the braking of the robot is not immediate. The effect of the intervention is to prevent the speed from increasing (`next(s)!=s+1`).

We saw in Section 2.2 (in Chapter 2) that only a non-permissive strategy can be synthesized for SI_I with the braking intervention only (see Figure 6.1 for reminder). We see that the intervention blocking the arm not only cuts the transitions from w_1 to `cata` and from w_3 to `cata`, but also from w_1 to w_2 and from w_3 to w_2: the state w_2 is not reachable.

An ideal intervention would only cut the transition to the catastrophic state(s). This would mean controlling exactly the evolution of all the variables on the transition (change, or freeze the variables values). However, the considered variables can be completely independent, therefore controlling them exactly simultaneously seems unrealistic. For example, the transition from w_1 (`s=1 & a=0`, i.e., the arm is folded while the speed is within the margin) to `cata` (`s=2 & a=1`, i.e., the arm is extended while the speed is higher than the maximum allowed value) can be cut if we prevent the arm extension and the increase in speed up to a value higher than $speed_{max}$. But also, one has to allow the speed to increase while the arm remains folded (`a=0`), or the arm to extend while the speed remains within the margin (the speed could fluctuate within the margin, but not cross the threshold). This seems to be more of a complex control function, than of an

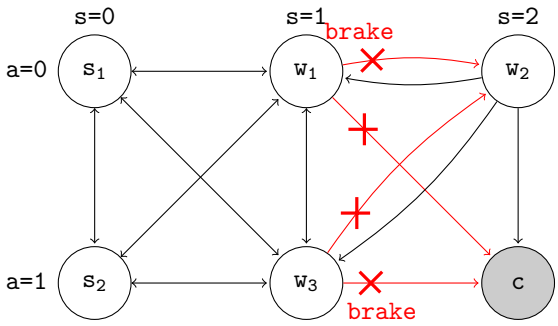

Figure 6.1: Non-permissive strategy synthesized for SI_I (repeated from 2.13)

intervention we can rely on. Generally, multi-variable interventions are hard to implement and thus not realistically trustworthy. Acknowledging this, we will only consider mono-variable interventions for the candidate interventions suggested for the synthesis. Note that several interventions can be applied in one state.

6.2 Identifying Candidate Interventions

The model limited to the safety invariant without a strategy, i.e., without the addition of intervention in warning states, is fully permissive. The addition of interventions in warning states cuts transitions not only to the catastrophic states but to warning and safe states. The interventions are real actions that the system can perform (e.g., triggering the brakes, prohibiting the movement of a part, etc), they can have a large impact. For instance, after an emergency stop, the system may not be able to move again without an intervention of the operator (there is no transition from this state to other safe states, the system is stuck). This results in a limitation of the system's possibilities of action, i.e., of the permissiveness.

In Chapter 4, we could identify which interventions (or combination of interventions) cut paths to some non-catastrophic states. We propose in this chapter to replace some of these interventions by more appropriate interventions (that would cut less or other transitions), in order to restore the permissiveness.

We propose an interactive approach, in which some new appropriate interventions can be suggested to the user, and reviewed by her. A new synthesis is then launched with the new interventions, potentially resulting in a fully permissive strategy.

6.2.1 Magical Interventions

The fact that the interventions cut more transitions than expected results in weakening permissiveness. Therefore the idea is to design interventions that, while being realistic (mono-variables, respecting the dynamics of the variable) will be as little restrictive as possible (cut as few transitions as possible). We will therefore privilege the intervention preventing a specific evolution of a variable than forcing one.

Warn. state	w_1: s=1 & a=0	w_2: s=2 & a=0	w_3: s=1 & a=1
Cata. state	c	c	c
Magical interv. (effect)	i(w_1-c): next(s)!=2 | next(a)!=1	i(w_2-c): next(s)!=2 | next(a)!=1	i(w_3-c): next(s)!=2 | next(a)!=1
Candidate interv. (effect)	i_{1,w_1}: next(s)!=2 i_{2,w_1}: next(a)!=1	i_{1,w_2}: next(s)!=2 i_{2,w_2}: next(a)!=1	i_{1,w_3}: next(s)!=2 i_{2,w_3}: next(a)!=1

Table 6.1: Identification of candidate interventions from the magical interventions for SI_I

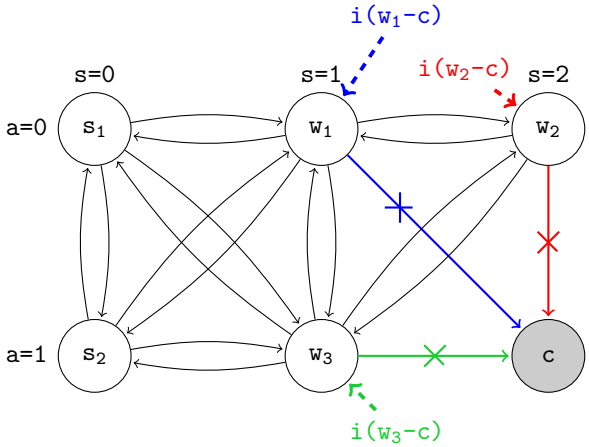

Figure 6.2: Strategy synthesized for SI_I with magical interventions

Theoretically, the only interventions needed in a warning state are interventions preventing the reachability of the neighboring catastrophic states and only these. We call these interventions *magical* as they may not realistically implementable. However, they guide us for the definition of new interventions.

Let us consider the invariant SI_I (the arm must not be extended when the speed is higher than $speed_{max}$). The warning states w_1, w_2 and w_3 would each need one magical intervention, cutting the transition to the catastrophic state c. Let us create three magical interventions i(w_1-c) that cuts $w_1 \rightarrow$ c, i(w_2-c) that cuts $w_2 \rightarrow$ c and i(w_3-c) that cuts $w_3 \rightarrow$ c. The effect of these interventions is described in Table 6.1 (the last line of the table is discussed later). As there is only one catastrophic state, all the interventions now have the same effect: preventing the variable from reaching the catastrophic values (i.e., their effect is !(next(s)=2 & next(a)=1)=next(s)!=1 | next(a)!=1). A strategy synthesized with these magical interventions would be as presented in Figure 6.2.

However, as mentioned above, we only want to consider mono-variable interventions. The effect of the magical intervention is written as a disjunctive normal form, i.e., a list of OR of clauses, a clause being here next(var)!=var_value. We can then split this multi-variable magical intervention into mono-variable interventions, each one having

Intervention name	Static precondition	Initial effect	Generalized effect
i_{1,w_1}	w_1	next(s)!=2	next(s)!=s+1
i_{2,w_1}	w_1	next(a)!=1	next(a)!=a+1
i_{1,w_2}	w_2	next(s)!=2	next(s)!=s
i_{2,w_2}	w_2	next(a)!=1	next(a)!=a+1
i_{1,w_3}	w_3	next(s)!=2	next(s)!=s+1
i_{2,w_3}	w_3	next(a)!=1	next(a)!=a

Table 6.2: Setting static precondition and generalizing the effects of the interventions for SI_I

for effect one of the clauses. The effect of corresponding mono-variables interventions for SI_I are shown in Table 6.1. We call these interventions *candidates* for the synthesis. The mono-variable interventions cut more transitions than the magical intervention, as they cut *all* the transitions along which a variable evolves in a certain way. For instance, an intervention preventing s from increasing, applied in w_1 cuts the transition from w_1 to the catastrophic state, but also the transition from w_1 to w_2 (see Figure 6.1).

The candidate interventions will be used for the tentative synthesis of a new strategy. However they may still cut too many transitions. It is still possible that no satisfying strategy is found with these new interventions.

6.2.2 Generalize the Interventions Effects

During the synthesis with SMOF, interventions are applied in warning state in order to synthesize a safe and permissive strategy. *Every* intervention is tried in *every* warning state and its validity evaluated with regards to the preconditions. It means that the more interventions are defined, the longer the synthesis will take to complete. We propose to reduce the number of candidate interventions: we add static preconditions to the interventions, re-write their effects and gather the ones that have the same effect.

Set static preconditions: In the previous section, we presented a way to identify a list of candidate interventions in analyzing the transition from each warning state to their neighboring catastrophic states. We therefore want the candidate interventions to only be applied in the warning states they have been designed for. For instance, the intervention i_{1,w_1} is designed for the warning state w_1, it must not be used in the warning state w_2 during the synthesis. To specify that, we add the warning state's characteristic predicate as a static precondition for the candidate intervention. The static precondition of the intervention i_{1,w_1} is w_1 (see Table 6.2).

Generalize the interventions effects: For variables defined as continuous, we can distinguish the effects precisely with regard to how the variable can evolve. Note that we use the term "continuous" as defined in SMOF: a discrete variable is said continuous if it can only increase by one, decrease by one or stay the same. For instance, the

Name	Static precondition	Effect	Sequential precondition
I_1	w_1 | w_3 = s=1	next(s)!=s+1	s=0
I_2	w_1 | w_2 = (s=1 | s=2) & a=0	next(a)!=a+1	TRUE
I_3	w_2 = s=2 & a=0	next(s)!=s	s=1
I_4	w_3 = s=1 & a=1	next(a)!=a	TRUE

Table 6.3: Resulting candidate interventions for SI_I

interventions i_{2,w_1}, i_{2,w_2} and i_{2,w_3} (see Table 6.1) have for effect `next(a)!=1`. However, the physical effect of these interventions is not the same depending on the state in which it is applied (the static precondition). The intervention i_{2,w_1}, applied in w_1: `s=1 & a=0`, has for effect to prevent the arm from extending, i.e., `next(a)!=a+1`. Similarly, i_{2,w_2} in w_2 prevents the extension of the arm. For i_{2,w_3} that is applied in w_3: `s=1 & a=1`, it specifies that the arm does not stay extended, i.e., the arm folds, expressed as `next(a)!=a`. Physically, this is a very different effect. It makes sense to differentiate it from the two others interventions.

We propose, for the interventions impacting continuous variables, to use the following pattern:

- `next(var)!= var-1;`
- `next(var)!= var;`
- `next(var)!= var+1.`

The generalization of the candidate interventions for SI_I is presented in Table 6.2.

Merge the interventions: Finally, the interventions that have the same effect can be gathered in a single definition. That is the case for the interventions i_{1,w_1} and i_{1,w_3} for instance. Their effect (`next(s)!=s+1`) can be merged, resulting in one intervention only. The static preconditions are also gathered: the intervention I_1 resulting from the merging has for static precondition the static preconditions of i_{1,w_1} and i_{1,w_3}. The results of the merging step for SI_I are presented in Table 6.3 (the last column of the table concerning sequential preconditions will be explained in the next section).

6.2.3 Interactive Review of the Interventions

In order to make the interventions as realistic as possible, i.e., respecting the system's abilities, the user can review them before launching the synthesis.

Set sequential preconditions: The main parameter than cannot be anticipated automatically is the presence of sequential preconditions. These represent the latency of some interventions. For instance, the braking intervention is not immediately effective as the robot has some inertia. Therefore, the sequential precondition for this intervention

is that the speed variable has just crossed the margin threshold: only in this case the brakes would be effective before the robot has the time to reach the catastrophic value.

The user can therefore review the candidate interventions to add, if possible, the sequential preconditions that would represent the actual characteristics of the system. For the invariant SI_I, the two interventions concerning the speed have a sequential precondition, due to the inertia of the platform. Controlling the arm however is here considered immediate, for illustration purposes. These sequential preconditions are presented in Table 6.3.

Remove non-implementable interventions: As a more drastic review of the list of interventions, the user can also remove any intervention which cannot be implemented, for instance because the concerned variables are not controllable.

Remove duplicates: Also, some candidate interventions could be duplicates of the ones initially defined. They can be removed from the list. For instance, $\texttt{I}_1$ is actually the braking intervention that was initially defined, it can therefore be removed from the list of additional candidates interventions. The original braking intervention remains and will be available for the synthesis.

The resulting list of candidate interventions, along with the braking intervention initially defined, is encoded as follows:

```
-- Interv(static precondition, sequential precondition, effect);
VAR I2:  Interv( ( s=1 | s=2 ) & a=0 , TRUE, next(a)!=a+1);
VAR I3:  Interv( s=2 & a=0, s=1, next(s)!=s);
VAR I4:  Interv( s=1 & a=1, TRUE, next(a)!=a);
VAR brake:  Interv(TRUE, s=0, next(s)!=s+1);
```

With these interventions one strategy respecting both the safety and permissiveness is synthesized for the invariant SI_I (represented in Figure 6.3). Let us recall that the `flag` of an intervention is the triggering condition.

```
-- STRATEGY #1
DEFINE flag_brake := s=1 & a=1 ;
DEFINE flag_I2 := s=1 & a=0 | s=2 & a=0 ;
DEFINE flag_I3 := FALSE ;
DEFINE flag_I4 := FALSE ;
```

We recognize here the strategy that was synthesized in the Chapter 2. Indeed, the intervention $\texttt{I}_2$ actually models exactly the intervention blocking the arm that we previously used. We were able to suggest an appropriate intervention.

Note that the reviewing by the user of the list of interventions is optional. If the user does not review the definition of the interventions, the synthesis may run for a longer amount of time, and return more strategies, that will have to be manually sorted out afterwards.

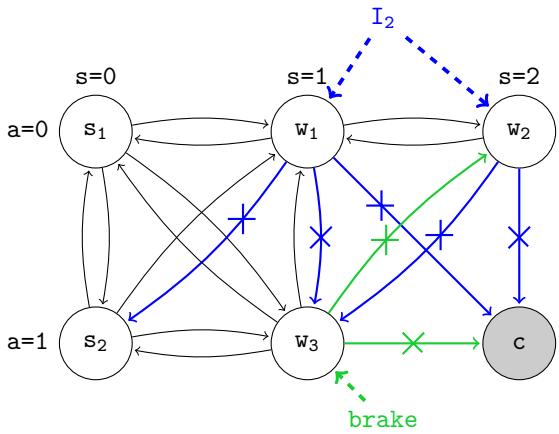

Figure 6.3: Strategies synthesized fore SI_I with the candidate interventions

6.3 Algorithm

In this section, we detail the algorithms used for the different steps of the suggestion of a list of candidate interventions for the synthesis with SMOF. How the different algorithms fit together is presented in Figure 6.4.

Creation of a partial strategy: In Chapter 4, we showed how to identify the interventions which have an impact on the permissiveness of a strategy (the permissiveness impact $\mathcal{PI}$ of the couples (state, intervention(s))). When a non-permissive strategy has been found, it can be used as an input for the suggestion of interventions. The interventions impacting the permissiveness (i.e., $i \in \mathcal{P}(I)$, there exists $s \in S \setminus S_c$, $\mathcal{PI}(s,i) \neq \emptyset$) are removed, resulting in a partial strategy. Some warning states are left with transitions to a catastrophic state, and the safety property is not satisfied. The suggestion of interventions therefore has to identify candidates interventions for the warning states with a transition to a catastrophic state only. The other warning states keep the interventions that they were associated with in the initial non-permissive strategy. This step is optional: if no initial strategy is chosen, the suggestion of interventions will design interventions for every warning state.

Let us consider the automaton $\mathcal{A}_{init}$ representing the system's behavior with the initial non-permissive strategy R_{init}. $\mathcal{A}_{init} = (S, T_{init}, s_0)$:

- S is the set of states, $S = S_s \cup S_w \cup S_c$:
  - S_s the set of safe states,
  - S_w the set of warning states (i.e., the states with a transition to a catastrophic state before synthesis of the strategy),
  - S_c the set of catastrophic states.
- T_{init} the transition function,

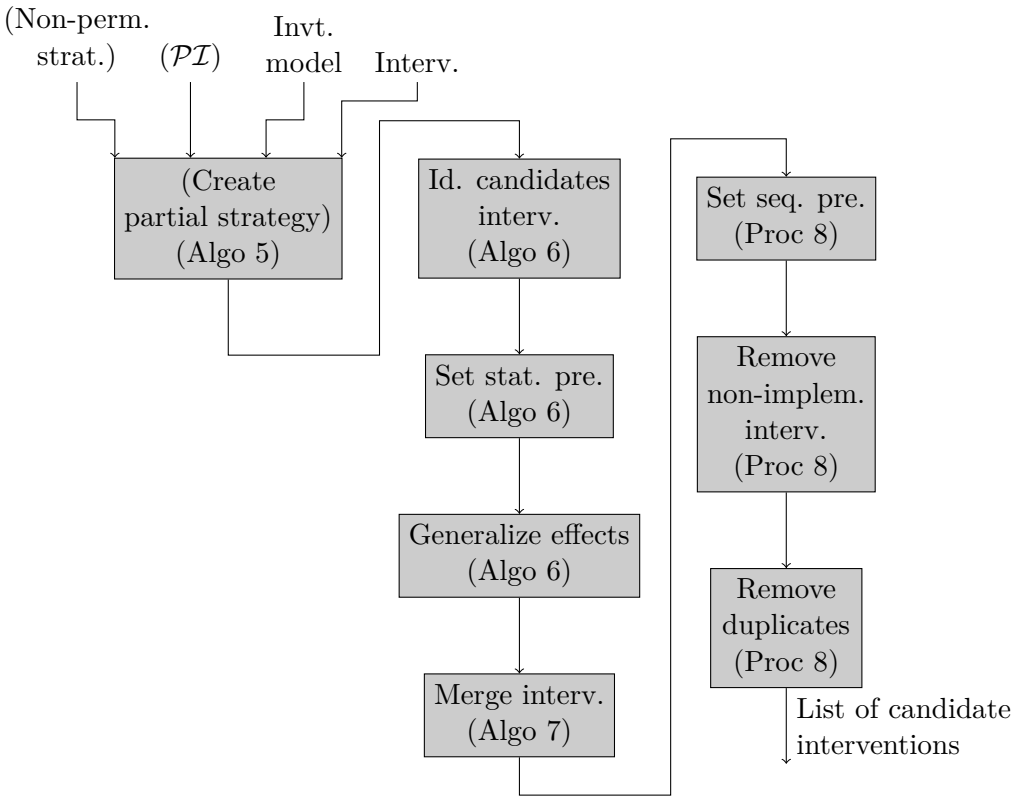

Figure 6.4: Process for the suggestion of interventions

- s_0 the initial state.

We call $\mathcal{V}$ the set of the n observable variables used to model the safety invariant. Each variable $v_j \in \mathcal{V}$ is defined on its domain D_j. A state $s \in S$ is a combination of variables and their valuations, i.e., $s \in D_1 \times ... \times D_n$. We write $v_j(s)$ the value of the variable v_j in the state s. We call $\mathcal{V}_c$ the set of continuous variables, $\mathcal{V}_c \subset \mathcal{V}$. The term continuous is used as defined in SMOF: the variable evolution is restricted to "the value increases of one", "the value decreases of one" and "the value stays the same". Such variable are defined using the `continuity` template in SMOF.

I is the set of interventions defined by the user. I can be empty if no initial strategy is chosen. In this case, every warning state is empty and has a transition to a catastrophic state, and the suggestion tool will design candidate interventions for all of them.

The initial non-satisfying strategy is a function $R_{init} : S_w \to \mathcal{P}(I)$ that associates to warning states zero, one or several interventions in I. R_{init} can be null: no intervention is associated to any warning state.

From R_{init}, we remove the interventions i_w from the states s_w in which $\mathcal{PI}(s_w, i_w) \neq \emptyset$ (i.e., the intervention i_w triggered in the state s_w contributes to cutting the paths to some non-catastrophic state(s)), following Algorithm 5.

Algorithm 5: Creating the partial strategy $R_{intermediary}$

Input: R_{init}, $\mathcal{A}_{init}$, $\mathcal{PI}$
$R_{intermediary} = \emptyset$
for $s_w \in S_w$ **do**
  for $i_w \in R_{init}(s_w)$ **do**
    if $\mathcal{PI}(s_w, i_w) = \emptyset$ **then**
      $R_{intermediary}(s_w) += i_w$
    end
  end
end
Output: $R_{intermediary}$

We now have $R_{intermediary} : S_w \to \mathcal{P}(I)$ (some states can be without interventions). This defines the automaton $\mathcal{A}_{intermediary} = (S, T_{intermediary}, s_0)$. This automaton represents the behavior with a partial strategy.

Design the list of candidate interventions: Let us consider the automaton $\mathcal{A}' = (S, T', s_0)$, being $\mathcal{A}_{intermediary}$ modified by a strategy R'. We want to find $R' : S_w \to \mathcal{P}(I')$, such that R' satisfies $Safe$ and $Perm$, the safety and permissiveness properties. SMOF automatically searches for R', given the set of interventions I'. We therefore want to find the set I' of candidate interventions for the synthesis of R'.

An intervention $i \in I'$ is defined through its static precondition, sequential precondition and effect. For simplified readability we will refer to the expressions of its components with $i(stat.\ pre.)$, $i(seq.\ pre.)$ and $i(effect)$.

The proposed algorithm is done in three steps. The first step is the identification of the appropriate candidate interventions for each warning state. It encompasses the

identification of their desired effects and static preconditions, and the generalization of the effects definitions for the continuous variables. It is detailed in Algorithm 6.

Algorithm 6: Identification of candidate interventions, setting of static preconditions and generalization.

Input: $\mathcal{A}_{intermediary} = (S, T_{intermediary}, s_0)$
1 i_{tab}=[ ] **for** $s_w \in S_w$, $\exists s_c \in S_c, (s_w, s_c) \in T_{intermediary}$ **do**
2 **for** $s_c \in S_c$, $(s_w, s_c) \in T_{intermediary}$ **do**
  `/* Identify the candidate interventions */`
3 **for** *j=0..Card(*$\mathcal{V}$*)* **do**
  `/* Effect */`
4 $i_{j,s_w}(effect) = next(v_j) \neq v_j(s_c))$
  `/* Static precondition */`
5 $i_{j,s_w}(stat.pre.) = s_w$
  `/* Generalize the definition for continuous variables */`
6 **if** $v_j \in \mathcal{V}_c$ **then**
7 **if** $v_j(s_w) < v_j(s_c)$ **then**
8 $i_{j,s_w}(effect) = next(v_j) \neq v_j + 1$
9 **end**
10 **if** $v_j(s_w) = v_j(s_c)$ **then**
11 $i_{j,s_w}(effect) = next(v_j) \neq v_j$
12 **end**
13 **if** $v_j(s_w) > v_j(s_c)$ **then**
14 $i_{j,s_w}(effect) = next(v_j) \neq v_j - 1$
15 **end**
16 **end**
17 $i_{tab} += i_{j,s_w}$
18 **end**
19 **end**
20 **end**
Output: i_{tab}

The second step merges all the intervention previously identified by their effect. This part is detailed in Algorithm 7.

The final step is the optional review of the interventions list by the user. It results in the final list of candidate interventions. It is presented in User Procedure 8.

The result of these algorithms is a list of interventions, defined through their static preconditions, sequential preconditions and effect. These interventions are then added to the invariant model, and the SMOF synthesis can be launched. We do not provide a guarantee that these interventions will be sufficient to find a safe and permissive strategy. Indeed, the candidate interventions are mono-variable and will therefore cut more transitions than just the one to the catastrophic state. However, these are the best realistic candidates.

Algorithm 7: Merging of the interventions

Input: i_{tab}

I=[ ], interv_checked=[], k=0
for $i \in i_{tab}$, $i \notin interv\_checked$ **do**
    /* Copy the intervention i in the list I */
    I[k]=i
    interv_checked+=i
    for $j \in i_{tab}$ **do**
        /* Add the precondition to the interv. in I with the same effect */
        if $j(effect) == i(effect)$ **then**
            $I[k](stat.pre.) = I[k](stat.pre.)|j(stat.pre.)$
            interv_checked+=j
        end
    end
    k++
end
I(stat.pre.)=Simplify(I(stat.pre.))

Output: I, the tentative list of candidate interventions

User Procedure 8: Interactive review of the list of interventions

Input: I, the tentative list of candidate interventions

$I_{final} = []$
/* The user removes the interventions that are not implementable */
$I = userRemoveCauseImplem(I)$
/* The user removes the interventions that are duplicate */
$I = userRemoveCauseDupl(I)$
for $i \in I$, **do**
    /* For the remaining interventions, the user sets sequential preconditions (if needed) */
    $i(seq.\ pre.) = userSetSeqPre(i)$
    $I_{final} += i$
end

Output: I_{final}, the final list of candidate interventions

6.4 Application to an Example

We consider the example of the invariant SI_{II} (a gripped box must not be tilted over an angle a_0). For this safety invariant, two observations are used: the angle of rotation of the arm `a` is a continuous variable on which a margin is taken (`a={low, within the margin, high}={0,1,2}`), and the position of the gripper, that is either closed empty, open or closed with a box (`g={closed empty, open, closed with box}={0,1,2}`). Accordingly, the catastrophic state is expressed as `cata: a=2 & g=2`. Three warning states exist:

w_0: `a = 1 & g = 1`
w_1: `a = 2 & g = 1`
w_2: `a = 1 & g = 2`

As presented in Chapter 2, one strategy was found for this invariant using two interventions, one braking the arm (prevent its rotation) and one locking the gripper (prevent it from closing). Let us assume now that no intervention was initially known. We want to see if we can suggest all the interventions necessary for a satisfying strategy. Note that we skip here the step of identifying a partial strategy from an initial non-permissive strategy, since we consider that no intervention has been defined.

Applying the algorithms 6 and 7 allows the identification of four candidate interventions, `I[0], I[1], I[2], I[3]`. It is detailed in Table 6.4. We can see the details of the loop iterations of Algorithm 7 (where the interventions with the same effect are merged) applied to the example.

These four interventions are added to the model, and the SMOF synthesis is run. Eight strategies are synthesized, that are both safe and permissive. This is already a decent number of strategies to go through to identify if one is implementable. The user, in this case, didn't review the list of interventions before launching the synthesis.

Interactive review of the interventions: Now let us see the strategies that would be found if the user reviews the list of candidate interventions. The intervention `I[0]` affects the angle of rotation of the arm: `I[0](effect)=next(a)!=a+1`, i.e., it prevents the rotation of the arm. There is some inertia on the arm's movement, therefore this intervention would only be efficient if applied when the threshold has just been crossed: `I[0](seq. pre.)= a=0`. This sequential precondition is added to the definition of `I[0]`. Similarly, `I[2]`, which can be applied in w_1, i.e., when the arm is fully rotated and the gripper open, has for effect to make the angle of rotation a change, in this case to make it decrease (as it is already at its maximum value). This also requires a sequential precondition due to the inertia: `I[2](seq. pre.)= a=1`. Concerning the interventions `I[1]` and `I[3]`, they have an effect on the gripper, that is considered to be immediate, no sequential precondition is needed. The reviewed interventions are detailed in Table 6.5.

Using this reviewed list of candidate interventions, the SMOF synthesis returns the two following strategies:

For all the warning states $s_w \in S_w$ that have transitions to catastrophic states		
w_0: a=1 & g=1	w_1: a=2 & g=1	w_2: a=1 & g=2
For all the neighboring catastrophic states of s_w		
cata	cata	cata
Identify the candidate interventions (Algo. 6)		
i_{0,w_0}(stat. pre.)=w_0 i_{0,w_0}(eff.)= next(a)!=2 i_{1,w_0}(stat. pre.)=w_0 i_{1,w_0}(eff.)=next(g)!=2	i_{0,w_1}(stat. pre.)=w_1 i_{0,w_1}(eff.)=next(a)!=2 i_{1,w_1}(stat. pre.)=w_1 i_{1,w_1}(eff.)=next(g)!=2	i_{0,w_2}(stat. pre)=w_2 i_{0,w_2}(eff.)=next(a)!=2 i_{1,w_2}(stat. pre.)=w_2 i_{1,w_2}(eff.)=next(g)!=2
Generalize the effects definition for continuous variables (Algo. 6)		
i_{0,w_0}(eff.)=next(a)!=a+1 i_{1,w_0}(eff.)=next(g)!=g+1	i_{0,w_1}(eff.)=next(a)!=a i_{1,w_1}(eff.)=next(g)!=g+1	i_{0,w_2}(eff.)=next(a)!=a+1 i_{1,w_2}(eff.)=next(g)!=g
Merge the interventions with the same effect (Algo. 7)		
I is empty First loop iteration: for i_{0,w_0} I[0](stat. pre.)=w_0; I[0](eff.)=next(a)!=a+1; Second loop iteration: for i_{1,w_0} i_{1,w_0}(eff.)!=I[0](eff.) I[1](stat. pre.)=w_0; I[1](eff.)=next(g)!=g+1; Third loop iteration: for i_{1,w_1} i_{1,w_0}(eff.)=I[1](eff.) I[1](stat. pre.)=w_0 | w_1 ; Fourth loop iteration: for i_{0,w_1} i_{0,w_1}(eff.)!=I[0](eff.), i_{0,w_1}(eff.)!=I[1](eff.) I[2](stat. pre.)=w_1; I[2](eff.)=next(a)!=a; Fifth loop iteration: for i_{0,w_2} i_{0,w_2}(eff.)=I[0](eff.) I[0](stat. pre.)=w_0 | w_2; Sixth loop iteration: for i_{1,w_2} i_{1,w_2}(eff.)!=I[0](eff.), i_{1,w_2}(eff.)!=I[1](eff.), i_{1,w_2}(eff.)!=I[2](eff.) I[3](stat. pre.)=w_2; I[3](eff.)=next(g)!=g;		
Tentative list of candidate interventions:		
I[0](stat. pre.)=w_0 | w_2 I[0](eff.)=next(a)!=a+1 I[1](stat. pre.)=w_0 | w_1 I[1](eff.)=next(g)!=g+1 I[2](stat. pre.)=w_1 I[2](eff.)=next(a)!=a I[3](stat. pre.)=w_2 I[3](eff.)=next(g)!=g		

Table 6.4: Algorithms 6 and 7 applied to the gripper invariant

Name	Static precondition	Sequential precondition	Effect
I[0]	w_0 | w_2	a=0	next(a)!=a+1
I[1]	w_0 | w_1	TRUE	next(g)!=g+1
I[2]	w_1	a=1	next(a)!=a
I[3]	w_2	TRUE	next(g)!=g

Table 6.5: Final list of candidate interventions for SI_{II}

```
-- STRATEGY #1
DEFINE flag_I0 := FALSE ;
DEFINE flag_I1 := a = 1 & g = 1 | a = 2 & g = 1 ;
DEFINE flag_I2 := FALSE ;
DEFINE flag_I3 := a = 1 & g = 2 ;

-- STRATEGY #2
DEFINE flag_I0 := a = 1 & g = 2 ;
DEFINE flag_I1 := a = 1 & g = 1 | a = 2 & g = 1 ;
DEFINE flag_I2 := FALSE ;
DEFINE flag_I3 := FALSE ;
```

The first strategy is quite counterintuitive. It uses the intervention I[1] in the states w_0 and w_1, i.e., decreases the angle of rotation whenever the gripper is open and the angle of rotation above the margin. It uses the intervention I[3] in the state w_2, i.e., it opens the gripper if the gripper is closed with a box and the arm rotated more than the margin. This strategy is indeed satisfying with regard to this invariant: the box is never tilted over the maximum angle, since it is no longer in the grip. However, this strategy implies to drop the box (I[3] in w_2), which is not satisfying with regard to another invariant (do not drop a box). If the user chose to keep this strategy, the consistency analysis would reveal a conflict with the invariant stating that a box must not be dropped.

The second strategy uses I[0] in w_2, i.e., prevents the rotation of the arm when a box is held, and uses I[1] in w_0 and w_1, i.e., prevents the gripper from closing (gripping a box) when the arm is tilted over the margin value. This strategy is the strategy that had been synthesized in the Chapter 2. We therefore validate that we were able to find the expected strategy without initially specifying any intervention.

The user had to review carefully the four interventions that were initially proposed by the tool, however this doesn't seem to be too burdensome. The number of intervention was limited. Also, if the user didn't want to review the interventions, they could review the 8 strategies synthesized with the initial candidate interventions: the satisfying strategy was also part of the set of strategy initially synthesized.

6.5 Conclusion

In this chapter, we propose to replace interventions that have an impact on the permissiveness by new ones. We propose an interactive approach for the design of these new interventions. A first definition of the candidate interventions can be automatically inferred from the invariant model, with or without a partial strategy. They are identified from a magical intervention that would cut only the transition to the catastrophic states. The candidate interventions are realistic as they respect the dynamics of the continuous variables, and are mono-variable, therefore avoiding a very fine joint control of independent variables.

Then, the user can review the interventions to adapt them to the physical characteristics of the system. This step is optional, but helps for the synthesis of more suitable strategies. The algorithms we designed rely on an exhaustive exploration of the transitions to the catastrophic states. They might not be appropriate for large models. An approach through heuristic search could be explored.

We saw that for our examples we were able to synthesize satisfying strategies using new interventions. However, further validation on larger case study would be necessary. Particularly, we have not been able to analyze cases where the variable are not all continuous or have a more complex definition of the continuity (as for the zones models of the collision invariant SI_4), therefore calling for different types of candidate interventions. It would be interesting to evaluate the approach on such a case.

Take Aways:

- We can improve non-permissive strategies by replacing some of their interventions (the ones with a nonzero permissiveness impact);
- The interventions can be designed automatically;
- The user can review the candidate interventions to make the synthesis more efficient;
- It is possible to synthesize appropriate strategies without prior knowledge on any of the available interventions.

Conclusion

Autonomous systems typically fulfill various missions and tasks, while evolving in unstructured and dynamic environments. They thus may have to face hazardous situations. To ensure their safety, monitoring is an appropriate technique: a safety monitor observes the system and reacts, should a danger be detected, keeping the system in a safe state. A safety monitor obeys to a set of safety rules, specifying the appropriate reaction to a detected violation. These rules must ensure that the monitor keeps the system safe (safety property), but also that it allows the system to perform its tasks (permissiveness properties). SMOF has been designed to synthesize safety rules for such monitors. It takes as input a model of the system's behavior, the available interventions and the desired safety and permissiveness properties. Then, its synthesis algorithm searches for a satisfying strategy, and, if successful, outputs a set of safe and permissive strategies. In this work we were interested in cases where SMOF fails to return a satisfying solution.

Contributions

We applied SMOF to a new case study —a maintenance robot that performs a light measurement task on airports runways— and revisited the results from a previous case study —a manufacturing robot. While doing so, we faced cases where SMOF cannot find a strategy. When that was the case, we had to manually search for the appropriate solution to implement to overcome this issue. This was a burdensome task and several iterations were needed. Three questions emerged when manually searching for a solution to SMOF synthesis' failure: how to understand why the synthesis fails? How to tune the permissiveness requirements to take into account the system's expected functionalities? And how to identify new candidate interventions for the synthesis? Our contributions answer these questions.

First, we developed a diagnosis tool, that helps the user identify why a strategy fails to fulfill some permissiveness requirements. We introduced two key concepts. The *permissiveness deficiencies* of a strategy qualify the permissiveness loss imposed by a strategy. The *permissiveness impact* of a couple (state, intervention(s)) points to the interventions that contribute to discard the paths to some operational states. These two parameters help the user to identify the blocking points for the synthesis, and making a decision on the changes to apply on the system or the requirements. This diagnosis tool has been successfully implemented in a Python version, and applied to several examples.

Second, we created a template for the user to adapt the permissiveness requirements to the system's expected functionalities. When the permissiveness properties are too drastic for the synthesis algorithm to find a solution, we also propose ways to weaken them with traceable impact on the functionalities. We can better qualify the tradeoff between safety and functionality due to the monitor. This second tool has also been implemented (in C++, for integration purposes with SMOF), and we were able to use it in conjunction with SMOF to find solutions when SMOF was returning no solution.

Third, we designed an interactive method to identify a list of interventions to be injected in the synthesis. The interventions are designed to be realistic and implementable, and the user can review them to adapt them to the system's actual abilities. These interventions can be used as a replacement to interventions having an impact on the permissiveness, or when no intervention is initially defined. Even if this approach has been formally described and successfully applied to examples, we did not implement the corresponding tool. This work was more prospective than for the two previous tools. However, no further step is needed for a successful implementation.

Our three contributions are an extension of the SMOF process, and have been successfully applied to examples extracted from two industrial case studies. We could now solve problems that could not previously be solved or needed some strong expertise.

This work can also be seen as a toolbox for solving cases when SMOF fails to return a satisfying solution. Each brick can be used alone or in combinations with the others, in different orders, depending on the case. Their addition to the SMOF process makes it more flexible and adaptable to a larger set of problems. For instance, we could handle cases where no intervention is initially declared. We can also imagine using these new bricks for the analysis and improvement of an existing monitor.

This research could be of interest to engineers in the robotics and autonomous systems industry who wish to improve their process of designing fault tolerance mechanisms. As much as possible, the use of formal methods and complex algorithms is hidden in our contributions, which makes them accessible for a user with little experience in these fields.

Limitations

Our work presents some limitations. We identify some of them here and mention ideas for improvement.

First, if most of the models are fairly simple with state spaces of a moderate size, the scalability of our contributions for larger models is an issue. Several of the algorithm designed in our work rely on an exhaustive check of a list of properties (for the diagnosis) or on the exhaustive exploration of the transitions to the catastrophic states (for the suggestion of interventions). These algorithms will not scale well for very large models. The computation time and needed resources may be excessive. To palliate this issue, the implementation of heuristic methods could be a valid approach. For instance, instead of exploring all the possible candidate interventions, we could identify interventions patterns from previous examples. These could be used in priority by the tool.

Second, the diagnosis tool that has been designed during this research takes as input a non-permissive strategy. It's goal is to analyze why this strategy is non-permissive (i.e., why the synthesis failed to synthesize a satisfying strategy) in order to identify how to improve it. However, the SMOF synthesis algorithm may return a large number of such non-permissive strategies. Choosing the initial strategy to be analyzed is a key point and can become complicated for the user when too many choices are available. It would be interesting to work on defining the best candidate strategy for the improve-

ments we propose (the replacement of some interventions, or the definition of custom permissiveness properties).

Perspectives

A short term perspective is the full implementation of the tools presented in this manuscript, including the suggestion of interventions facility. An important effort is needed to increase the usability of this toolbox in order to be used by engineers, which may raise issues about the combination of the different tools. For instance, for a given non-permissive strategy it would be possible to tune the permissiveness with the custom permissiveness tool and to identify new interventions with the suggestion of interventions tool. An interface needs to be built between these two tools to ease their use in combination. Also, the user interface needs to be improved, allowing for a better interactivity.

In our approach, we do not consider the cost of the interventions regarding the impact on autonomy for instance (other definitions of cost may be used). On real systems, some interventions may be more "expensive" (i.e., with a high cost) than others. For instance, an emergency brake, requiring an intervention of the operator will degrade autonomy more than a soft brake, after which the robot can start moving again. A slight deviation of trajectory to avoid an obstacle may be preferred to a full stop. To take this into account, a cost value could be associated to the interventions. The synthesis would then search for the cheapest safe and permissive strategy.

While taking into account the cost of an intervention, the synthesis could also consider its probability of success. Some interventions may have a probability of success lower than one. For instance, a trajectory calculation function can be use to avoid an obstacle, or to bring the system away from a dangerous zone. However, such a function is typically implemented on the main control board, where most resources are available. It cannot be fully trusted, which is often justified by a calculation of an integrity level of a safety function (as in ISO61508 standard [IEC 61508-1, 2010]). For instance, some faults on the control board —that cannot be fully verified due to its complexity— could propagate to this safety function, making it fail. The intervention will have a probability of success lower than one. Such interventions could then be associated to interventions with a guaranteed effect. We would then construct multi-level strategies: an intervention with a probability of success lower than one is triggered; if it fails, a guaranteed intervention is triggered. Considering interventions with a low probability of success is interesting because interventions with a probability of success lower than one typically have a low cost (e.g., replan and execute a new safe trajectory cannot be guaranteed but preserves system autonomy), when the contrary is true for guaranteed interventions (e.g., emergency stop is often implemented with high integrity but autonomy is then fully degraded). The search for a strategy then needs to satisfy an objective function such as the cost is minimal while guaranteeing that no catastrophic state can be reached. This would imply in our approach to include a quantification that could be discrete or even probabilistic. In this latter case, we would have then to switch to prob-

abilistic model-checkers. We have started to explore the idea of using non-guaranteed interventions in collaboration with the research team MECS at KTH, Stockholm, for an autonomous vehicle [Svensson et al., 2018]. We designed a new intervention, called a *safe stop trajectory planner* (SSTP), that brings the vehicle to a safe stop area (typically outside of the active traffic lane) by selecting among a set of pre-computed trajectories. This intervention is not guaranteed. It can fail if the context does not allow to find a suitable trajectory among the pre-computed set. It has then been associated to two others interventions, an intervention that makes the system slow down (i.e., increasing the probability for the SSTP to find a suitable trajectory) and an emergency stop, that guaranteed that the system stops in case no safe trajectory can be executed.

Following an idea similar to the cost of the interventions, the functionalities specified with the custom permissiveness template may not all have the same weight or priority. For instance, a robot movement functionality may be preferred over a manipulation functionality. If the synthesis fails to satisfy all the permissiveness properties associated with the functionalities, it may have to give up on some of them. It would then be interesting to associate a priority level to the functionalities. The synthesis could then search for the strategy satisfying the permissiveness associated with the functionalities with the highest priority. Integrating this feature in the existing toolset would be easily feasible. The custom permissiveness tool can automatically restrict functionalities. The notion of priority would just have to be added to guide the tool in the choice of the restriction to apply. However, this raises the question of how to identify the appropriate functionalities to model, and how to associate the right priority level to them. One possible direction would be to use model-based system descriptions of tasks and functionalities as we have done using HAZOP-UML for analysis hazards.

Bibliography

[Abdessalem et al., 2018] Abdessalem, R. B., Nejati, S., Briand, L. C., and Stifter, T. (2018). Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pages 1016–1026, Gothenburg, Sweden. ACM Press.

[Alami et al., 1998] Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An Architecture for Autonomy. *The International Journal of Robotics Research*, 17(4):315–337.

[Alemzadeh et al., 2015] Alemzadeh, H., Chen, D., Lewis, A., Kalbarczyk, Z., Raman, J., Leveson, N., and Iyer, R. (2015). Systems-Theoretic Safety Assessment of Robotic Telesurgical Systems. In Koornneef, F. and van Gulijk, C., editors, *Computer Safety, Reliability, and Security*, volume 9337, pages 213–227. Springer International Publishing, Cham.

[Alexander and Arnold, 2013] Alexander, R. and Arnold, J. (2013). Testing Autonomous Robot Control Software Using Procedural Content Generation. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Bitsch, F., Guiochet, J., and Kaâniche, M., editors, *Computer Safety, Reliability, and Security*, volume 8153, pages 33–44, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Alexander et al., 2009] Alexander, R., Kelly, T., and Herbert, N. (2009). Deriving Safety Requirements for Autonomous Systems. *4th SEAS DTC Technical Conference*.

[Ames et al., 2015] Ames, A. D., Tabuada, P., Schürmann, B., Ma, W.-L., Kolathaya, S., Rungger, M., and Grizzle, J. W. (2015). First steps toward formal controller synthesis for bipedal robots. In *Proceedings of the 18th International Conference on Hybrid Systems Computation and Control - HSCC '15*, pages 209–218, Seattle, Washington. ACM Press.

[Aniculaesei et al., 2016] Aniculaesei, A., Arnsberger, D., Howar, F., and Rausch, A. (2016). Towards the Verification of Safety-critical Autonomous Systems in Dynamic Environments. *Electronic Proceedings in Theoretical Computer Science*, 232:79–90.

[Arora et al., 2015] Arora, S., Choudhury, S., Althoff, D., and Scherer, S. (2015). Emergency maneuver library - ensuring safe navigation in partially known environments. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6431–6438, Seattle, WA, USA. IEEE.

[Askarpour et al., 2016] Askarpour, M., Mandrioli, D., Rossi, M., and Vicentini, F. (2016). SAFER-HRC: Safety Analysis Through Formal vERification in Human-Robot

Collaboration. In Skavhaug, A., Guiochet, J., and Bitsch, F., editors, *Computer Safety, Reliability, and Security*, volume 9922, pages 283–295. Springer International Publishing, Cham.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Basu et al., 2006] Basu, A., Bozga, M., and Sifakis, J. (2006). Modeling Heterogeneous Real-time Components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12, Pune, India. IEEE.

[Bensalem et al., 2009] Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I., and Thanh-Hung, N. (2009). Designing autonomous robots. *IEEE Robotics & Automation Magazine*, 16(1):67–77.

[Blanquart et al., 2004] Blanquart, J.-P., Fleury, S., Hernek, M., Honvault, C., Ingrand, F., Poncet, T., Powell, D., Strady-Lécubin, N., and Thévenod, P. (2004). Software Safety Supervision On-board Autonomous Spacecraft. In *Proceedings of the 2nd European Congress Embedded Real Time Software (ERTS'04)*, page 9.

[Bloem et al., 2015] Bloem, R., Könighofer, B., Könighofer, R., and Wang, C. (2015). Shield Synthesis: Runtime Enforcement for Reactive Systems. In Baier, C. and Tinelli, C., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 533–548. Springer Berlin Heidelberg.

[Bruyninckx, 2001] Bruyninckx, H. (2001). Open robot control software: the OROCOS project. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 3, pages 2523–2528 vol.3.

[Bruyninckx et al., 2013] Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. (2013). The BRICS component model: a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*, page 1758, Coimbra, Portugal. ACM Press.

[Böhm and Gruber, 2010] Böhm, P. and Gruber, T. (2010). A Novel HAZOP Study Approach in the RAMS Analysis of a Therapeutic Robot for Disabled Children. In Schoitsch, E., editor, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 15–27. Springer Berlin Heidelberg.

[Casimiro et al., 2014] Casimiro, A., Rufino, J., Pinto, R. C., Vial, E., Schiller, E. M., Morales-Ponce, O., and Petig, T. (2014). A kernel-based architecture for safe cooperative vehicular functions. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pages 228–237, Pisa. IEEE.

[Cavalli et al., 2003] Cavalli, A., Gervy, C., and Prokopenko, S. (2003). New approaches for passive testing using an Extended Finite State Machine specification. *Information and Software Technology*, 45(12):837–852.

[Cook et al., 2014] Cook, D., Vardy, A., and Lewis, R. (2014). A survey of AUV and robot simulators for multi-vehicle operations. In *2014 IEEE/OES Autonomous Underwater Vehicles (AUV)*, pages 1–8.

[CPSE-Labs, 2018] CPSE-Labs (2018). CPSE Labs - Cyber-Physical Systems Engineering Labs. http://www.cpse-labs.eu/index.php. Accessed on 2018-07-20.

[Crestani et al., 2015] Crestani, D., Godary-Dejean, K., and Lapierre, L. (2015). Enhancing fault tolerance of autonomous mobile robots. *Robotics and Autonomous Systems*, 68:140–155.

[Delgado et al., 2004] Delgado, N., Gates, A., and Roach, S. (2004). A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872.

[Desai et al., 2017] Desai, A., Dreossi, T., and Seshia, S. A. (2017). Combining Model Checking and Runtime Verification for Safe Robotics. In Lahiri, S. and Reger, G., editors, *Runtime Verification*, Lecture Notes in Computer Science, pages 172–189. Springer International Publishing.

[Dhillon and Fashandi, 1997] Dhillon, B. and Fashandi, A. (1997). Safety and reliability assessment techniques in robotics. *Robotica*, 15(6):701–708.

[Dixon et al., 2014] Dixon, C., Saunders, J., Webster, M., Fisher, M., and Dautenhahn, K. (2014). "The Fridge Door is Open"–Temporal Verification of a Robotic Assistant's Behaviours. In Leonardis, A., Mistry, M., Witkowski, M., and Melhuish, C., editors, *Advances in Autonomous Robotics Systems*, volume 8717, pages 97–108. Springer International Publishing, Cham.

[Dogramadzi et al., 2014] Dogramadzi, S., Giannaccini, M. E., Harper, C., Sobhani, M., Woodman, R., and Choung, J. (2014). Environmental Hazard Analysis - a Variant of Preliminary Hazard Analysis for Autonomous Mobile Robots. *Journal of Intelligent & Robotic Systems*, 76(1):73–117.

[Durand et al., 2010] Durand, B., Godary-Dejean, K., Lapierre, L., Passama, R., and Crestani, D. (2010). Fault tolerance enhancement using autonomy adaptation for autonomous mobile robots. In *2010 Conference on Control and Fault-Tolerant Systems (SysTol)*, pages 24–29, Nice, France. IEEE.

[Ertle et al., 2010] Ertle, P., Gamrad, D., Voos, H., and Soffker, D. (2010). Action planning for autonomous systems with respect to safety aspects. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 2465–2472, Istanbul, Turkey. IEEE.

[Falcone et al., 2012] Falcone, Y., Fernandez, J.-C., and Mounier, L. (2012). What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382.

[Falcone et al., 2013] Falcone, Y., Havelund, K., and Reger, G. (2013). A Tutorial on Runtime Verification. *Engineering dependable software systems*, page 35.

[Falzon and Pace, 2013] Falzon, K. and Pace, G. J. (2013). Combining Testing and Runtime Verification Techniques. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Machado, R. J., Maciel, R. S. P., Rubin, J., and Botterweck, G., editors, *Model-Based Methodologies for Pervasive and Embedded Software*, volume 7706, pages 38–57. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Feth et al., 2017] Feth, P., Schneider, D., and Adler, R. (2017). A Conceptual Safety Supervisor Definition and Evaluation Framework for Autonomous Systems. In Tonetta, S., Schoitsch, E., and Bitsch, F., editors, *Computer Safety, Reliability, and Security*, volume 10488, pages 135–148. Springer International Publishing, Cham.

[Fiacre, 2018] Fiacre (2018). The Fiacre language and Frac compiler Home Page by LAAS/CNRS. http://projects.laas.fr/fiacre/. Accessed on 2018-11-13.

[Fisher et al., 2013] Fisher, M., Dennis, L., and Webster, M. (2013). Verifying autonomous systems. *Communications of the ACM*, 56(9):84.

[Foughali et al., 2018] Foughali, M., Berthomieu, B., Dal Zilio, S., Hladik, P.-E., Ingrand, F., and Mallet, A. (2018). Formal Verification of Complex Robotic Systems on Resource-Constrained Platforms. In *FormaliSE: 6th International Conference on Formal Methods in Software Engineering*, Gothenburg, Sweden.

[Fox and Das, 2000] Fox, J. and Das, S. (2000). *Safe and sound - Artificial Intelligence in Hazardous Applications*. AAAI Press - The MIT Press.

[Gainer et al., 2017] Gainer, P., Dixon, C., Dautenhahn, K., Fisher, M., Hustadt, U., Saunders, J., and Webster, M. (2017). CRutoN: Automatic Verification of a Robotic Assistant's Behaviours. In Petrucci, L., Seceleanu, C., and Cavalcanti, A., editors, *Critical Systems: Formal Methods and Automated Verification*, Lecture Notes in Computer Science, pages 119–133. Springer International Publishing.

[Goldberg et al., 2005] Goldberg, A., Havelund, K., and McGann, C. (2005). Runtime verification for autonomous spacecraft software. In *2005 IEEE Aerospace Conference*, pages 507–516, Big Sky, MT, USA. IEEE.

[Gribov and Voos, 2014] Gribov, V. and Voos, H. (2014). A multilayer software architecture for safe autonomous robots. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, Barcelona, Spain. IEEE.

[Gspandl et al., 2012] Gspandl, S., Podesser, S., Reip, M., Steinbauer, G., and Wolfram, M. (2012). A dependable perception-decision-execution cycle for autonomous robots. In *2012 IEEE International Conference on Robotics and Automation*, pages 2992–2998, St Paul, MN, USA. IEEE.

[Guiochet, 2016] Guiochet, J. (2016). Hazard analysis of human–robot interactions with HAZOP–UML. *Safety Science*, 84:225–237.

[Haddadin et al., 2011] Haddadin, S., Suppa, M., Fuchs, S., Bodenmüller, T., Albu-Schäffer, A., and Hirzinger, G. (2011). Towards the Robotic Co-Worker. In Pradalier, C., Siegwart, R., and Hirzinger, G., editors, *Robotics Research*, Springer Tracts in Advanced Robotics, pages 261–282. Springer Berlin Heidelberg.

[HAZOP-UML, 2018] HAZOP-UML (2018). Hazard Identification with HAZOP and UML. https://www.laas.fr/projects/HAZOPUML/. Accessed on 2018-08-02.

[Horányi et al., 2013] Horányi, G., Micskei, Z., and Majzik, I. (2013). Scenario-based automated evaluation of test traces of autonomous systems. In *SAFECOMP 2013-Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, page NA.

[Huang et al., 2014] Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., and Rosu, G. (2014). ROSRV: Runtime Verification for Robots. In Bonakdarpour, B. and Smolka, S. A., editors, *Runtime Verification*, volume 8734, pages 247–254. Springer International Publishing, Cham.

[Huber et al., 2017] Huber, M., Rizaldi, A., Keinholz, J., Feldle, J., Immler, F., Althoff, M., Hilgendorf, E., and Nipkow, T. (2017). Formalising and Monitoring Traffic Rules for Autonomous Vehicles in Isabelle/HOL. In Polikarpova, N. and Schneider, S., editors, *Integrated Formal Methods*, volume 10510, pages 50–66. Springer International Publishing, Cham.

[IEC 61508-1, 2010] IEC 61508-1 (2010). IEC 61508-1:2010 functional safety, smart city.

[Ing-Ray Chen, 1997] Ing-Ray Chen (1997). Effect of parallel planning on system reliability of real-time expert systems. *IEEE Transactions on Reliability*, 46(1):81–87.

[ISO 12100, 2013] ISO 12100 (2013). Safety of machinery — General principles for design — Risk assessment and risk reduction.

[ISO/IEC Guide 51, 2014] ISO/IEC Guide 51 (2014). Safety aspects – Guidelines for their inclusion in standards.

[Jiang et al., 2017] Jiang, H., Elbaum, S., and Detweiler, C. (2017). Inferring and monitoring invariants in robotic systems. *Autonomous Robots*, 41(4):1027–1046.

[Kane et al., 2015] Kane, A., Chowdhury, O., Datta, A., and Koopman, P. (2015). A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System. In Bartocci, E. and Majumdar, R., editors, *Runtime Verification*, Lecture Notes in Computer Science, pages 102–117. Springer International Publishing.

[Kane and Koopman, 2013] Kane, A. and Koopman, P. (2013). Ride-through for Autonomous Vehicles. In *SAFECOMP 2013-Workshop CARS (2nd Workshop on Critical Automotive applications: Robustness & Safety) of the 32nd International Conference on Computer Safety, Reliability and Security*.

[Klein, 1991] Klein, P. (1991). The Safety-Bag Expert System in the Electronic Railway Interlocking System ELEKTRA. In Zarri, G. P., editor, *Operational Expert System Applications in Europe*, pages 1–15. Pergamon.

[Knightscope, 2016] Knightscope (2016). Knightscope Issues Field Incident Report. https://www.businesswire.com/news/home/20160713006532/en/Knightscope-Issues-Field-Incident-Report. Accessed on 2018-11-27.

[Knightscope, 2017] Knightscope (2017). Knightscope Issues MIN42 Field Incident Report. https://www.businesswire.com/news/home/20170728005099/en/Knightscope-Issues-MIN42-Field-Incident-Report. Accessed on 2018-11-27.

[KUKA, 2018] KUKA (2018). industrial intelligence 4.0_beyond automation. https://www.kuka.com/en-de. Accessed on 2018-07-20.

[Kwiatkowska et al., 2007] Kwiatkowska, M., Norman, G., and Parker, D. (2007). Stochastic Model Checking. In Bernardo, M. and Hillston, J., editors, *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, Lecture Notes in Computer Science, pages 220–270. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Lesire et al., 2012] Lesire, C., Doose, D., and Cassé, H. (2012). Mauve: a Component-based Modeling Framework for Real-time Analysis of Robotic Applications. In *7th full day Workshop on Software Development and Integration in Robotics (ICRA2012 - SDIR VII)*.

[Leucker and Schallhart, 2009] Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303.

[Ligatti et al., 2009] Ligatti, J., Bauer, L., and Walker, D. (2009). Run-Time Enforcement of Nonsafety Policies. *ACM Transactions on Information and System Security*, 12(3):1–41.

[Lopes et al., 2016] Lopes, Y. K., Trenkwalder, S. M., Leal, A. B., Dodd, T. J., and Groß, R. (2016). Supervisory control theory applied to swarm robotics. *Swarm Intelligence*, 10(1):65–97.

[Lotz et al., 2011] Lotz, A., Steck, A., and Schlegel, C. (2011). Runtime monitoring of robotics software components: Increasing robustness of service robotic systems. In *2011 15th International Conference on Advanced Robotics (ICAR)*, pages 285–290, Tallinn, Estonia. IEEE.

[Luckcuck et al., 2018] Luckcuck, M., Farrell, M., Dennis, L., Dixon, C., and Fisher, M. (2018). Formal Specification and Verification of Autonomous Robotic Systems: A Survey. *arXiv:1807.00048 [cs]*. arXiv: 1807.00048.

[Lussier et al., 2007] Lussier, B., Gallien, M., Guiochet, J., Ingrand, F., Killijian, M.-O., and Powell, D. (2007). Fault Tolerant Planning for Critical Robots. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 144–153, Edinburgh, UK. IEEE.

[Machin, 2015] Machin, M. (2015). *Synthèse de règles de sécurité pour des systèmes autonomes critiques*. phdthesis.

[Machin et al., 2015] Machin, M., Dufosse, F., Guiochet, J., Powell, D., Roy, M., and Waeselynck, H. (2015). Model-Checking and Game theory for Synthesis of Safety Rules. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pages 36–43, Daytona Beach Shores, FL. IEEE.

[Machin et al., 2018] Machin, M., Guiochet, J., Waeselynck, H., Blanquart, J.-P., Roy, M., and Masson, L. (2018). SMOF: A Safety Monitoring Framework for Autonomous Systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(5):702–715.

[Mallet et al., 2010] Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., and Ingrand, F. (2010). GenoM3: Building middleware-independent robotic components. In *2010 IEEE International Conference on Robotics and Automation*, pages 4627–4632, Anchorage, AK. IEEE.

[Malm et al., 2010] Malm, T., Viitaniemi, J., Latokartano, J., Lind, S., Venho-Ahonen, O., and Schabel, J. (2010). Safety of Interactive Robotics—Learning from Accidents. *International Journal of Social Robotics*, 2(3):221–227.

[Masson et al., 2017] Masson, L., Guiochet, J., Waeselynck, H., Desfosses, A., and Laval, M. (2017). Synthesis of Safety Rules for Active Monitoring: Application to an Airport Light Measurement Robot. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, pages 263–270, Taichung, Taiwan. IEEE.

[Muscettola et al., 2002] Muscettola, N., Dorais, G. A., Fry, C., Levinson, R., Plaunt, C., and Clancy, D. (2002). IDEA: Planning at the Core of Autonomous Reactive Agents. In *AIPS Workshop on On-line Planning and Scheduling*.

[NTBS, 2016] NTBS (2016). Collision Between a Car Operating With Automated Vehicle Control Systems and a Tractor-Semitrailer Truck Near Williston, Florida, May 7, 2016. Technical report.

[NuSMV, 2018] NuSMV (2018). NuSMV home page. http://nusmv.fbk.eu/. Accessed on 2016-07-19.

[O'Brien et al., 2014] O'Brien, M., Arkin, R. C., Harrington, D., Lyons, D., and Jiang, S. (2014). Automatic Verification of Autonomous Robot Missions. In Brugali, D., Broenink, J. F., Kroeger, T., and MacDonald, B. A., editors, *Simulation, Modeling, and Programming for Autonomous Robots*, Lecture Notes in Computer Science, pages 462–473. Springer International Publishing.

[Pace et al., 2000] Pace, C., Seward, D., and Sommerville, I. (2000). A Safety Integrated Architecture for an Autonomous Excavator. In *International Symposium on Automation and Robotics in Construction*.

[Pathak et al., 2013] Pathak, S., Pulina, L., Metta, G., and Tacchella, A. (2013). Ensuring safety of policies learned by reinforcement: Reaching objects in the presence of obstacles with the iCub. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 170–175, Tokyo. IEEE.

[Pecheur, 2000] Pecheur, C. (2000). Verification and Validation of Autonomy Software at NASA.

[Pettersson, 2005] Pettersson, O. (2005). Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88.

[Pike et al., 2012] Pike, L., Niller, S., and Wegmann, N. (2012). Runtime Verification for Ultra-Critical Systems. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Khurshid, S., and Sen, K., editors, *Runtime Verification*, volume 7186, pages 310–324. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Powell et al., 2012] Powell, D., Arlat, J., Chu, H. N., Ingrand, F., and Killijian, M. (2012). Testing the Input Timing Robustness of Real-Time Control Software for Autonomous Systems. In *2012 Ninth European Dependable Computing Conference*, pages 73–83, Sibiu. IEEE.

[Py and Ingrand, 2004] Py, F. and Ingrand, F. (2004). Dependable execution control for autonomous robots. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 2, pages 1136–1141, Sendai, Japan. IEEE.

[Quigley et al., 2009] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). ROS : an open-source Robot Operating System. In *International Conference on Robotics and Automation (ICRA), Workshop on open source software*.

[Ramadge and Wonham, 1987] Ramadge, P. J. and Wonham, W. M. (1987). Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230.

[Roderick et al., 2004] Roderick, S., Roberts, B., Atkins, E., and Akin, D. (2004). The Ranger robotic satellite servicer and its autonomous software-based safety system. *IEEE Intelligent Systems*, 19(5):12–19.

[SAPHARI, 2018] SAPHARI (2018). SAPHARI - Safe and Autonomous Physical Human-Aware Robot Interaction - Home. http://www.saphari.eu/. Accessed on 2018-07-20.

[Shoaei et al., 2010] Shoaei, M. R., Lennartson, B., and Miremadi, S. (2010). Automatic generation of controllers for collision-free flexible manufacturing systems. In *2010 IEEE International Conference on Automation Science and Engineering*, pages 368–373, Toronto, ON. IEEE.

[SMOF, 2018] SMOF (2018). SMOF : Safety MOnitoring Framework. https://www.laas.fr/projects/smof/. Accessed on 2018-08-10.

[SMT-LIB, 2018] SMT-LIB (2018). SMT-LIB The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu/. Accessed on 2018-08-15.

[Sorin et al., 2016] Sorin, A., Morten, L., Kjeld, J., and Schultz, U. P. (2016). Rule-based Dynamic Safety Monitoring for Mobile Robots. 7(1):120–141.

[Sotiropoulos et al., 2016] Sotiropoulos, T., Guiochet, J., Ingrand, F., and Waeselynck, H. (2016). Virtual Worlds for Testing Robot Navigation: A Study on the Difficulty Level. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 153–160, Gothenburg. IEEE.

[Sotiropoulos et al., 2017] Sotiropoulos, T., Waeselynck, H., Guiochet, J., and Ingrand, F. (2017). Can Robot Navigation Bugs Be Found in Simulation? An Exploratory Study. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 150–159, Prague, Czech Republic. IEEE.

[Sterela, 2018] Sterela (2018). Sterela : Advanced Solution Maker. http://www.sterela.fr/en/. Accessed on 2018-07-20.

[Stringfellow et al., 2010] Stringfellow, M., Leveson, N., and Owens, B. (2010). Safety-Driven Design for Software-Intensive Aerospace and Automotive Systems. *Proceedings of the IEEE*, 98(4):515–525.

[Svensson et al., 2018] Svensson, L., Masson, L., Mohan, N., Ward, E., Brenden, A. P., Feng, L., and Törngren, M. (2018). Safe Stop Trajectory Planning for Highly Automated Vehicles: An Optimal Control Problem Formulation. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 517–522.

[Tina, 2018] Tina (2018). The TINA toolbox Home Page - TIme petri Net Analyzer - by LAAS/CNRS. http://projects.laas.fr/tina//. Accessed on 2018-11-13.

[Tomatis et al., 2003] Tomatis, N., Terrien, G., Piguet, R., Burnier, D., Bouabdallah, S., Arras, K., and Siegwart, R. (2003). Designing a secure and robust mobile interacting robot for the long term. In *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, volume 3, pages 4246–4251, Taipei, Taiwan. IEEE.

[Troubitsyna and Vistbakka, 2018] Troubitsyna, E. and Vistbakka, I. (2018). Deriving and Formalising Safety and Security Requirements for Control Systems. In Hoshi, M. and Seki, S., editors, *Developments in Language Theory*, volume 11088, pages 107–122. Springer International Publishing, Cham.

[Tuleap, 2018] Tuleap (2018). Tuleap • Open Source Agile Project Management and Software Development tools. https://www.tuleap.org/. Accessed on 2018-08-27.

[Täubig et al., 2012] Täubig, H., Frese, U., Hertzberg, C., Lüth, C., Mohr, S., Vorobev, E., and Walter, D. (2012). Guaranteeing functional safety: design for provability and computer-aided verification. *Autonomous Robots*, 32(3):303–331.

[van Nunen et al., 2016] van Nunen, E., Tzempetzis, D., Koudijs, G., Nijmeijer, H., and van den Brand, M. (2016). Towards a safety mechanism for platooning. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 502–507, Gotenburg, Sweden. IEEE.

[Vistbakka et al., 2018] Vistbakka, I., Majd, A., and Troubitsyna, E. (2018). Multi-layered Approach to Safe Navigation of Swarms of Drones. In Gallina, B., Skavhaug, A., Schoitsch, E., and Bitsch, F., editors, *Computer Safety, Reliability, and Security*, volume 11088 of *Lecture Notes in Computer Science*, pages 112–125. Springer International Publishing.

[Volpe et al., 2001] Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., and Das, H. (2001). The CLARAty architecture for robotic autonomy. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, volume 1, pages 1/121–1/132 vol.1.

[Wagner et al., 2008] Wagner, S., Eusgeld, I., Kroger, W., and Guaglio, G. (2008). Balancing safety and availability for an electronic protection system. In *European Safety and Reliability Conference*.

[Woodman et al., 2012] Woodman, R., Winfield, A. F., Harper, C., and Fraser, M. (2012). Building safer robots: Safety driven control. *The International Journal of Robotics Research*, 31(13):1603–1626.

[Zaman et al., 2013] Zaman, S., Steinbauer, G., Maurer, J., Lepej, P., and Uran, S. (2013). An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In *2013 IEEE International Conference on Robotics and Automation*, pages 482–489, Karlsruhe, Germany. IEEE.

[Zou et al., 2014] Zou, X., Alexander, R., and McDermid, J. (2014). Safety Validation of Sense and Avoid Algorithms Using Simulation and Evolutionary Search. In Bondavalli, A. and Di Giandomenico, F., editors, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 33–48. Springer International Publishing.

Résumé : Un moniteur de sécurité actif est un mécanisme indépendant qui est responsable de maintenir le système dans un état sûr, en cas de situation dangereuse. Il dispose d'observations (capteurs) et d'interventions (actionneurs). Des règles de sécurité sont synthétisées, à partir des résultats d'une analyse de risques, grâce à l'outil SMOF (Safety MOnitoring Framework), afin d'identifier quelles interventions appliquer quand une observation atteint une valeur dangereuse. Les règles de sécurité respectent une propriété de sécurité (le système reste das un état sûr) ainsi que des propriétés de permissivité, qui assurent que le système peut toujours effectuer ses tâches.

Ce travail se concentre sur la résolution de cas où la synthèse échoue à retourner un ensemble de règles sûres et permissives. Pour assister l'utilisateur dans ces cas, trois nouvelles fonctionnalités sont introduites et développées. La première adresse le diagnostique des raisons pour lesquelles une règle échoue à respecter les exigences de permissivité. La deuxième suggère des interventions de sécurité candidates à injecter dans le processus de synthèse. La troisième permet l'adaptation des exigences de permissivités à un ensemble de tâches essentielles à préserver. L'utilisation des ces trois fonctionnalités est discutée et illustrée sur deux cas d'étude industriels, un robot industriel de KUKA et un robot de maintenance de Sterela.
Mots clés : Moniteur de sécurité, Systèmes autonomes, Méthodes formelles

Abstract: An active safety monitor is an independent mechanism that is responsible for keeping the system in a safe state, should a hazardous situation occur. Is has observations (sensors) and interventions (actuators). Safety rules are synthesized from the results of the hazard analysis, using the tool SMOF (Safety MOnitoring Framework), in order to identify which interventions to apply for dangerous observations values. The safety rules enforce a safety property (the system remains in a safe state) and some permissiveness properties (the system can still perform its tasks).

This work focuses on solving cases where the synthesis fails to return a set of safe and permissive rules. To assist the user in these cases, three new features are introduced and developed. The first one addresses the diagnosis of why the rules fail to fulfill a permissiveness requirement. The second one suggests candidate safety interventions to inject into the synthesis process. The third one allows the tuning of the permissiveness requirements based on a set of essential functionalities to maintain. The use of these features is discussed and illustrated on two industrial case studies, a manufacturing robot from KUKA and a maintenance robot from Sterela.
Key-words: Safety monitoring, Autonomous systems, Formal methods