Specifying Safety Monitors for Autonomous Systems Using Model-Checking

Mathilde Machin[1,2], Fanny Dufossé[1,2], Jean-Paul Blanquart[3], Jérémie Guiochet[1,2], David Powell[1,2], and Hélène Waeselynck[1,2]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France
firstname.lastname@laas.fr
[3] Airbus Defence and Space, 31 rue des cosmonautes, 31402 Toulouse, France
jean-paul.blanquart@astrium.eads.net

Abstract. Autonomous systems operating in the vicinity of humans are critical in that they potentially harm humans. As the complexity of autonomous system software makes the zero-fault objective hardly attainable, we adopt a fault-tolerance approach. We consider a separate safety channel, called a monitor, that is able to partially observe the system and to trigger safety-ensuring actuations. A systematic process for specifying a safety monitor is presented. Hazards are formally modeled, based on a risk analysis of the monitored system. A model-checker is used to synthesize monitor behavior rules that ensure the safety of the monitored system. Potentially excessive limitation of system functionality due to presence of the safety monitor is addressed through the notion of permissiveness. Tools have been developed to assist the process.

Keywords: Safety Monitoring, Safety Rules, Autonomous Robotics.

1 Introduction

Autonomous systems such as robots and unmanned vehicles are widely studied and technically feasible. An important bottleneck for their effective deployment in human environments is the safety concerns of both users and certification authorities. Various ad-hoc safety measures have been designed, often focused on particular risks, such as collision. However, if autonomous systems are to be certified, the method needs to be generalized. We propose here a general method to build high-level safety specifications based on hazard analysis.

The autonomous systems of interest to us offer a wide range of features and operate in a diverse unstructured environment. They can thus be complex, which makes them difficult to verify. Moreover, diversity of the environment implies that testing cannot significantly cover the situations that the system will face. Here, we choose a classical fault tolerance approach by considering online safety measures implemented in a device called a *safety monitor*, that is simple and independent from the main control channel, and thus easier to verify. The monitor is solely responsible for safe system behavior. To this end, the monitor is

A. Bondavalli and F. Di Giandomenico (Eds.): SAFECOMP 2014, LNCS 8666, pp. 262–277, 2014.

equipped with means for context observation (i.e., sensors) and is able to trigger safety interventions. The monitor behavior is specified declaratively by a set of *safety rules*, each defining one intervention to apply in certain observation conditions. However, safety interventions may also prevent the system from fulfilling its functions. For instance, a vehicle whose emergency brakes are permanently engaged is useless. We require the monitor to be *permissive* with respect to the possibility for the system to perform useful tasks.

Continuing the work of Mekki-Mokhtar *et al.* [1], we propose a process based on hazard analysis to specify safety monitors and extend it by means of formal methods. Once a hazard is identified, it is necessary to specify what the monitor has to do to avoid it, i.e., the safety rules. We aim to explore solutions very early in the autonomous system design process. Thus, many observations and interventions can be considered in a first design iteration, whereas only the most appropriate ones are actually developed and implemented. We propose to use model-checking to explore and check the specifications.

The main contributions of this paper are:

- A method to explore possible safety specifications by using model-checking.
- A method for modeling permissiveness in temporal logic.
- A set of tools to support the methodology[1].

First, we present the overall concepts and process in Section 2. Section 3 details the exploration of possible safety rules in a discrete model, which is applied in Section 4 to a mobile manufacturing robot. Related work is discussed in Section 5 and Section 6 presents conclusions and future work.

2 Baseline and Concepts

We introduce here the underlying concepts of our work, based on definitions adapted from [1], and then present the overall process.

2.1 Concepts

Taking inspiration from the IEC 61508 standard [2], we define a *safety monitor* as a device responsible for safety, in opposition to the main control channel which is responsible for all other functional and non-functional requirements of the system. The monitor is equipped with means for context observation (i.e., sensors) and able to trigger safety interventions. The safety monitor is independent from the main control channel, as regards its means of observation, computation and intervention. It is required to protect against all faults that adversely affect safety, including interaction faults. The whole safety channel is assumed fault-free (for example, we consider that the sensors available to the monitor are perfect, without uncertainty.) In practice, this must be achieved through classical redundancy and verification techniques. We focus in our work on the upstream task of obtaining a correct high-level specification with respect to safety and permissiveness.

[1] Available at `http://webhost.laas.fr/TSF/archives/safety_rule_synthesis`

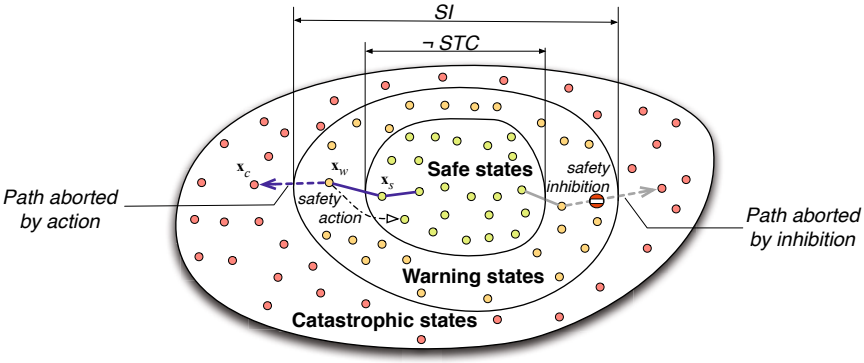

Fig. 1. Partition of system states in catastrophic, warning and safe states

A safety invariant (SI) is a necessary and sufficient condition to avoid a hazardous situation. If a safety invariant is violated, we assume that damage is immediate and irreversible, with no possible recovery. We refer to any state violating the safety invariant as a *catastrophic state*.

Example: "the robot speed shall not exceed 3 m/s*" (where* 3 m/s *is the speed beyond which harm is considered to be inevitable).*

A safety intervention is an activity carried out explicitly to prevent the system from violating a safety invariant by constraining the system behavior. An intervention is only applicable in states satisfying its associated *precondition*. We distinguish two types of interventions: inhibitions and actions.

A safety inhibition prevents a change in system state. When triggered, an inhibition is assumed to be immediately effective.

Example : "lock the wheels" (with "robot stationary" as precondition).

A safety action triggers a change in system state (and implicitly prevents other state changes).

Example : "apply emergency brake".

A safety trigger condition (STC) is a condition that, when asserted, triggers a safety intervention. The intervention is applied when the STC is true. The STC is chosen such that it becomes true before the safety invariant is violated.

Example: "the robot speed is greater than 2 m/s *(i.e., less than the safety invariant threshold of* $3m/s$*)".*

A safety rule defines a way of behaving in response to a hazardous situation. A safety rule can be operationalized as an if-then rule:

Safety rule $\triangleq$ *if* [safety trigger condition] *then* [safety intervention].

Example: "if the robot speed is greater than 2 m/s *then apply emergency brake."*

As illustrated in Figure 1, the safety invariant defines the partition between catastrophic states and non-catastrophic states of the monitored system. Interventions have to be applied before the catastrophe, i.e., in non-catastrophic

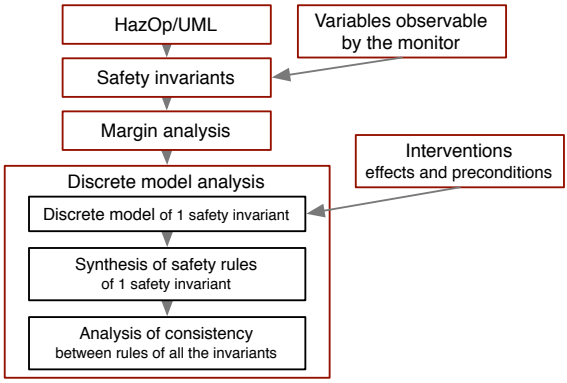

Fig. 2. Overview of the process

states. Now, interventions add constraints to the system behavior. So the set of non-catastrophic states is partitioned into warning states, where interventions are applied, and safe states, in which the system operates without constraint. The warning states are defined such that every path from a safe state (e.g., $\mathbf{x}_s$ on Figure 1) to a catastrophic state, e.g., $\mathbf{x}_c$, passes through a warning state, e.g., $\mathbf{x}_w$. The warning state enables triggering of an intervention to abort the path to the catastrophic state.

We assess the monitor and its safety rule set according to the following three properties:

Safety is the ability to ensure that the safety invariants are never violated, i.e., that catastrophic states are unreachable.

Permissiveness is the ability to allow the system to perform its tasks.

Validity specifies that no intervention is applied while its precondition is false.

Safety and permissiveness are antagonistic. We take this antagonism into account by designing the monitor to be *maximally permissive with respect to safety*, i.e., to restrict functionality only to the extent necessary to ensure safety.

2.2 Process Overview

Figure 2 presents the overall process. We base our process on a HAZOP-UML hazard analysis, which outputs safety invariants expressed in natural language. We consider as a running example a mobile robot with a manipulator arm and the informal safety invariant *The arm must not be extended beyond the base when the speed is greater than* V_0.

The safety invariant is then expressed formally with predicates on variables that are observable by the monitor. We focus for now only on predicates involving a variable compared to a fixed threshold. This type of safety threshold is amenable to formal verification and is used in many real systems. Considering the two monitor observations: the absolute speed v, and a Boolean observation

of the arm position a ($true$ when the arm is above the base, $false$, when the arm is extended), the example safety invariant is formalized as $v < V_0 \vee a = true$.

The margin analysis partitions non-catastrophic states into safe states and warning states by splitting variable value intervals or sets. This is done one variable after another. For example, the speed interval $[0, V_0[$ from the safety invariant is partitionable according to a margin m in two intervals $[0, V_0 - m[$ and $[V_0 - m, V_0[$. In the case of arm position, the observation is Boolean. The singleton value set $\{true\}$ cannot be partitioned, hence no margin exists. Formal conditions for the existence of a margin are studied in [1].

From the margin analysis, we can discretize variables involved in the safety invariant in order to synthesize safety rules. We call this the discrete model analysis, which is detailed in Section 3. It is composed of three main steps: creation of a discrete model, rule synthesis, and rule consistency checking. In order to keep models simple enough to be validated, each safety invariant is modeled separately. The state variables of the model are the observable variables discretized by intervals according to the thresholds of the safety invariant and the existing margins. The discrete model (e.g., Figure 3) is the Cartesian product of the variable partitions. A catastrophic state is one that violates the safety invariant (there is one catastrophic state on Figure 3, labeled C). The warning states (W) are those that lead the system to the catastrophe in one step. Interventions are modeled using the same discretized variables. In the example the monitor is able to brake (action) and to prevent the arm from extending (inhibition).

The monitor is responsible for neutralizing every transition leading to a catastrophic state. For instance, Figure 4 illustrates a satisfying safety rule set, which applies braking in $s3$ and arm inhibition in $s1$ and $s2$. Additionally to the transitions leading directly to the catastrophic state, several other transitions are deleted. The safety rule set respects the safety properties, as the system cannot enter the catastrophic state. All non-catastrophic states are reachable. Nevertheless, there is some loss of permissiveness as the system cannot stay in $s3$. We consider this to be acceptable. In Sections 3.4 and 3.5, we propose two methods to find systematically such safety rule sets.

As safety invariants are processed separately, the final step is to check the consistency between the safety rule sets from different safety invariants. This is addressed in Section 3.6.

3 Discrete Model Analysis

Given a safety invariant, several safety rules are usually needed to avoid violation of the safety invariant. We call a *safety strategy* a set of rules applied with respect to a single safety invariant. In this section, we aim to synthesize a safe, permissive and valid strategy based on the discrete model.

We propose two approaches to synthesize strategies (Figure 5). The *automatic method* finds strategies fast, given permissiveness requirements, by exploring automatically the various combinations of safety rules. The *interactive method* enables the user to adapt permissiveness requirements, and to build or modify a strategy rule by rule.

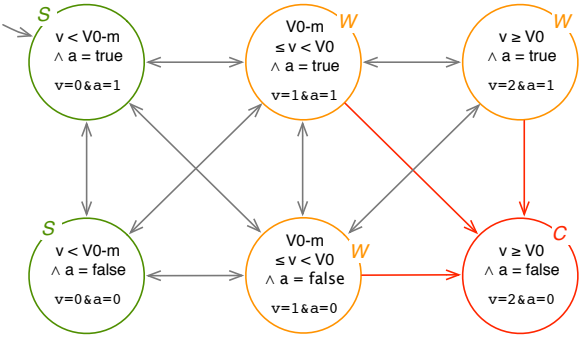

Fig. 3. The example discrete model from the partitions $\{true, false\}$ for arm position, and $\{[0, V_0 - m[, [V_0 - m, V_0[, [V_0, V_{max}[\}$ for speed

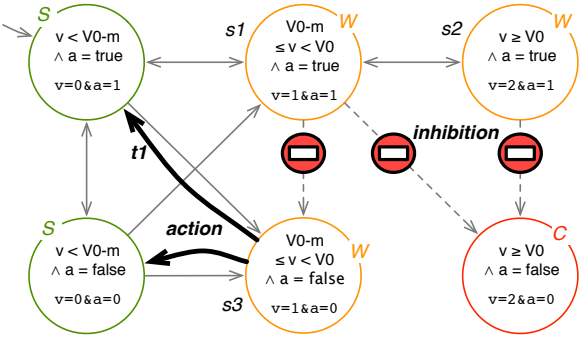

Fig. 4. The example model with a safety rule set

The interactive method is used whenever there is no solution to the given model and requirements. It informs the user on how to adapt the submitted problem. The automatic method can then be used on the new problem to find all possible strategies

3.1 Tools

We use the modeling language SMV and the model-checker NuSMV2 [3]. SMV enables the declaration of integer variables and constraints on their behavior. NuSMV builds transparently the Cartesian product of the ranges of all variables. When no constraint is declared, all the combinations of variable values (i.e., states) are possible and all transitions between each pair of states are implicitly declared. Constraints are then added to delete undesired states and transitions. As for variables, time is discrete. It is modeled by the operator `next()`. NuSMV is well-adapted to our variable-oriented modeling approach. Moreover, the implicit transition declaration is convenient for modeling the whole physically possible behavior.

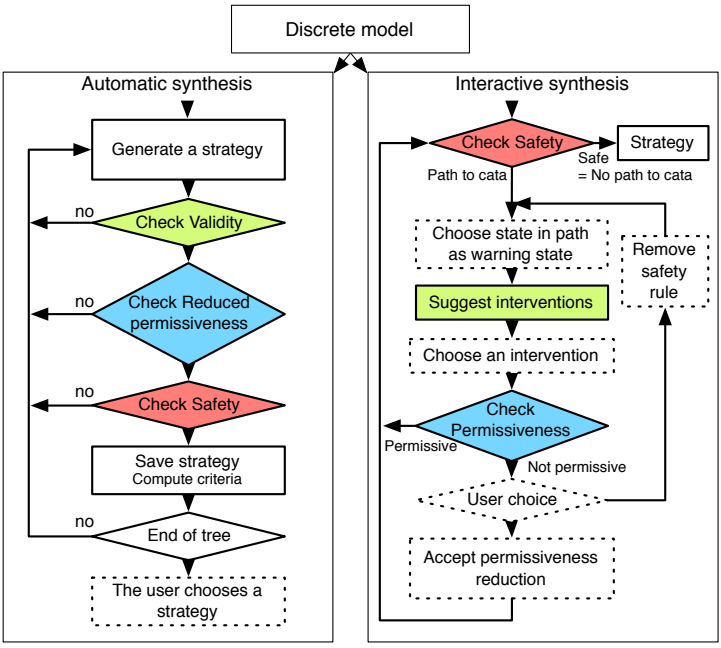

Fig. 5. The two methods for safety strategy synthesis

In the following, SMV code and output of NuSMV are given in typewriter font. We have developed a template file to facilitate the modeling and to allow the process to be automated.

3.2 System and Intervention Modeling

The domain of each variable of the safety invariant is partitioned according to the thresholds of the safety invariant and the margin (if it exists), and the resulting elements are numbered. For instance $\{[0, V_0 - m[,\ [V_0 - m, V_0[,\ [V_0, V_{max}[\}$ is encoded as {0,1,2} (see Figure 3). Continuity of variables, i.e., contiguity of partition elements, is modeled as the constraint: `next(x) = x | x+1 | x-1`, i.e., a variable `x` can stay in the same interval or move to an adjacent interval, but it cannot jump from one interval to another that has no common boundary.

We then model possible dependencies between variables. Nevertheless, some dependencies cannot be modeled in a discrete way or with a given partition. If a dependency is not modeled, the discrete model has less constraints than it should, or from another point of view, it has too many transitions. If this "super-graph" is safe, so is the "true" model. On the contrary, the permissiveness results of the super-graph are not trustworthy. The resulting strategies are always safe; but their level of permissiveness depends on the dependency modeling effort.

Interventions are always effective (when their preconditions are true), provided some environmental and dimensioning assumptions. A safety braking action

requires to consider for example a maximum slope rate, a maximum torque from the motors. Safety interventions are then modeled as constraints that may be applied or not. Consider a discretized speed `v`. The braking action and the acceleration inhibition can be modeled by:

```
braking ->  ( (v!=0 -> next(v)=v-1) & (v=0 -> next(v)=0) )
acc_inhibition -> next(v)!=v+1
```

As these examples show, an intervention usually adds a constraint on only one variable, and leaves the others free. For example, at the same time step: speed can be decreased by braking, and the arm can fold (as in transition $t1$ in Figure 4).

We make no restrictive assumption about the behavior of the main control channel. The system model represents what is physically possible in the system without a monitor. Therefore, safety interventions only remove transitions, i.e., possible behaviors, and cannot add transitions, i.e., add physically impossible behaviors.

Unlike classical model-checking, an integer value does not model a single interval width of an observable variable. Consequently, the time step has no determined value. The `next` operator models an elastic future, which can be very close or far away.

3.3 Safety, Permissiveness and Validity Modeling

Monitor properties are expressed in CTL (*Computation Tree Logic*), which is entirely supported by NuSMV without any syntax change. Time along paths is modeled by three operators: X for a property to hold in the next state, G to hold on the entire path, F to hold eventually. The branching aspect is modeled by A, all the branches, and E, there exists a branch. A CTL operator is composed of one branching operator and one time operator. It is applied on states, or more generally on statements on the system state.

To model safety, we use the atomic property *cata* to denote the catastrophic states. *cata* is the negation of the safety invariant, e.g., `cata := speed=2 & arm_pos=0`. Safety is modeled as the unreachability of the catastrophic states, i.e., in CTL, $AG \neg cata$. The expression of *cata* is the only user task in the initial property modeling. Permissiveness and validity properties are generated automatically. During the synthesis, the user is supposed to remove some permissiveness properties according to the accepted permissiveness loss choices.

Permissiveness is translated by three liveness properties applied to each non-catastrophic state s_{nc}:

- Simple reachability $EF\ s_{nc}$
  The state is reachable from the initial state.
- Universal reachability $AG\ EF\ s_{nc}$
  The state is reachable from any reachable state.
- Continuous (and universal) reachability $AG\ EF\ (s_{nc} \wedge EG\ s_{nc})$
  The state is reachable and the automaton can stay (indefinitely) in this state. If an action is applied to the state, the system cannot stay in the state. It is only a transient state, so the system cannot carry out tasks in this state.

Continuous reachability is stronger than universal reachability, which is stronger than simple reachability. The three properties checked separately on each warning state enable permissiveness to be assessed in a more detailed way than with a single binary value. It is usually impossible to obtain safety without some loss of permissiveness, and particularly with respect to continuous reachability.

It is possible but highly unlikely for variables to change their values simultaneously and independently. Such changes are called *diagonal transitions* by reference to the two variable case (cf. Figure 3). As permissiveness should not depend on such unlikely transitions, we choose to ignore diagonal transitions when checking permissiveness. *diag* denotes that the immediately fired transition was a "diagonal" transition. $mem(diag)$ is the memorization of *diag*, i.e., *diag* and $mem(diag)$ are initially false and as soon as *diag* is true, $mem(diag)$ becomes true and stays true even if the value of *diag* changes. To ignore diagonal transitions during permissiveness checking, the properties are modified using $mem(diag)$ as follows:

- Simple reachability $EF(\, s_{nc} \wedge \neg mem(diag))$
  The state is reachable by a path that always satisfies $\neg diag$, i.e., a path that has no simultaneous value changes of independent variables.
- Universal reachability $AG\Big(\neg mem(diag) \rightarrow EF\Big(s_{nc} \wedge \neg mem(diag)\Big)\Big)$
  The implication selects the part of the model without diagonal transitions and checks the reachability property only in this part. From the previous simple reachability property, we already know that this part is non-void.
- Continuous (and universal) reachability
  $AG\Big(\neg mem(diag) \rightarrow EF\Big(s_{nc} \wedge \neg mem(diag) \wedge EG\big(s_{nc}\big)\Big)\Big)$

The automaton without any safety rule is usually permissive because it is only a structure without specified behavior. Variables can change freely their values. Similarly, it is unsafe, as catastrophic states are reachable.

Validity specifies that interventions are not applied in states that violate their preconditions. We express this as:

$$AG \bigwedge_{i \in Interventions} i \rightarrow precondition_i$$

where $Interventions$ is the set of the candidate interventions and $precondition_i$ is the precondition associated to intervention i.

Once the safety invariant and the interventions have been defined, and the properties have been generated, we can synthesize a strategy using either the interactive method (Section 3.4) or the automatic method (Section 3.5).

3.4 Interactive Method

The interactive method (right side of Figure 5) uses the command-line interface of NuSMV and alias commands. The model-checker finds a path to a catastrophic state as a counter-example to the safety property. The user chooses

a warning state in this path to apply an intervention. The warning state is by default the state immediately preceding the catastrophic state. Then the model-checker determines whether each intervention is *locally* relevant. To this end, the warning state is declared as the initial state and additional properties, called *suggestion* properties, are checked. Suggestion properties are of the form $i \rightarrow (precondition_i \wedge AX \neg cata)$ where i is the intervention. The intervention is suggested if its precondition is satisfied (i.e., the rule is valid) and if it renders the catastrophic state unreachable in one step. If the state immediately preceding the catastrophic state is not suitable (e.g., no intervention can be applied), the user chooses an earlier state. When the nth state before the catastrophic state is selected, the property has to be modified to apply the AX operator n times to check that the catastrophic state is unreachable in n steps.

The user chooses an intervention among those suggested. The model-checker checks the permissiveness of the system with the new safety rule.

This is done iteratively until there are no more paths to catastrophic states, i.e., the system is safe. The selected safety rules constitute a satisfying strategy.

If the permissiveness test returns false, the user has three choices: accept the loss of permissiveness; try another intervention; or try another warning state. Selecting an earlier warning state in the path implies that downstream states will be unreachable, which negatively impacts permissiveness. It may however be relevant if a combination of safety rules makes many states unreachable.

The interactive method enables the user to customize in what states and to what extent permissiveness is required. But exploration can be slow, which is the reason why we have also developed an automatic method.

3.5 Automatic Method

The automatic method (left side of Figure 5) runs on the same model. It outputs all safe and valid strategies that satisfy the permissiveness requirements (if any such strategies exist). If full permissiveness is required, no result is obtained. On the contrary, the lower the requirements, the more results there are. We thus consider by default only simple and universal reachability, and compute criteria that help the user to choose. When there is no solution, the interactive method enables the user to find the blocking point and locally reduce the permissiveness requirement. The automatic method is then run with customized requirements.

The automatic method is based on the enumeration of the strategies through a branch-and-cut algorithm and the verification of properties by NuSMV [4]. The method is implemented using NuSMV scripts and a C program.

3.6 Consistency between Strategies

Different strategies may apply interventions simultaneously, which may be incompatible, e.g., braking and acceleration. To check strategy consistency, the previous models (with their strategies) are merged into a single model. When observable variables are common to several models but with different domain partitions, a new domain partition is defined by taking the union of the thresholds from the different models.

There are two types of inconsistency. For example, braking and acceleration impose incompatible constraints on speed, so the model-checker cannot compute a next state. This type of inconsistency is detected by a basic command. Other inconsistencies are not visible in the model because they cannot be modeled with the chosen partition or there is no impact on an observable variable. In these cases, we propose to list concurrent interventions to enable an expert to determine inconsistencies.

4 Case Study

Our case study is part of the SAPHARI (*Safe and Autonomous Physical Human-Aware Robot Interaction*) project [5]. The robot is composed of a mobile base and an articulated arm. It is an industrial co-worker in a manufacturing setting. It takes and places part boxes on shelves, work stations, or on the robot base in order to convey them. It operates in the human workspace. We study here two safety invariants from this robot.

4.1 Human/Arm Collision during Base Motion

Collision avoidance of the base trajectory relies only on base-to-obstacle distance sensor. Consequently, if the arm is unfolded and extends beyond the base during base motion, a collision between the arm and a human is possible. A very slow base movement is tolerated. This case is the same as the example of Section 2.2. The safety invariant is: *The arm must not be extended beyond the base when the base is moving (with speed higher than V_0).*

Discrete Model. The available observations are: 1) a Boolean observation of the arm position a; 2) linear absolute base speed v (to simplify we ignore rotation speed). a and v are independent. The safety invariant is formalized as $a = true \vee v < V_0$. The considered interventions are: 1) braking (of base wheels); 2) inhibit the arm motion to prevent it from extending beyond the base, this is possible only when the arm is above the base. A margin exists for the speed. The following excerpt of the SMV module encodes the discrete model of Figure 3. No other template modification is required.

```
MODULE Collision_SI
VAR -- Variable declarations
--Continuity(low bound, high bound, initial value, mode)
base_speed : Continuity(0,2,0,mode);
    -- 0:<V0-m,  1:V0-m<v<V0,  2:>V0
arm_pos : Continuity(0,1,1,mode);
    -- 1:above the base, 0:extended beyond

DEFINE cata:= (base_speed.v=2 & arm_pos.v=0);

VAR -- Intervention declarations
```

```
--Interv(precondition, flag to apply interv, effect, mode)
brake_base : Interv( base_speed.v!=0, flag_brake_base, next
    (base_speed.v)=base_speed.v - 1, mode);
inhib_arm : Interv( arm_pos.v=1, flag_inihb_arm, next(
    arm_pos.v)=1, mode);
```

The effect of `brake_base` is to decrease the speed. However, when `speed.v=0`, decreasing it violates the variable range, so `speed!=0` is set as precondition.

Interactive Method. We apply the algorithm of the right side of Figure 5. Checking for safety returns a path to the catastrophic state. The state immediately preceding the catastrophic state, chosen as a warning state, is defined by `base_speed.v = 1 & arm_pos.v = 0`. With this state as initial, suggestion properties are checked. The only suggested intervention is `brake_base`. We thus define its trigger `flag_brake_base` in a safety rule:

```
DEFINE flag_brake_base := base_speed.v=1 & arm_pos.v=0;
```

Permissiveness properties are true except for the continuous reachability of the warning state. This is expected since braking is an action intervention.

Another path to the catastrophic state results in defining `base_speed.v = 1 & arm_pos.v = 1` as a warning state. Both possible interventions are suggested. The inhibition `inhib_arm` is chosen since it does not decrease permissiveness:

```
DEFINE flag_inhib_arm := base_speed.v=1 & arm_pos.v=1;}
```

We check that permissiveness is indeed unchanged.

A third path to catastrophe defines the warning state `base_speed.v = 2 & arm_pos.v = 1` where both interventions are again suggested. We choose `inhib_arm` again and therefore add the warning state to `flag_inhib_arm`.

```
DEFINE flag_inhib_arm := (base_speed.v=1 & arm_pos.v=1) | (
    base_speed.v=2 & arm_pos.v=1);}
```

Checking for safety now returns true. The strategy so defined is valid, safe, and acceptably permissive. It is the same strategy as in Figure 4.

Automatic Method. From the same model `Collision_SI` the automatic method returns three strategies. Among the three generated strategies, two have two non-continuously reachable states and the last has only one such state. To minimize loss of permissiveness we choose the strategy with only one non-continuously reachable state.

```
 STRATEGY #2
--Criteria
non continuously reachable states 1
states with intervention 3
states with combined interv 0
total nb of interv 3
interv_brake_base used in 1 states
```

```
interv_inhib_arm used in 2 states
--Strategy definition
DEFINE flag_brake_base := flag_cinterv_1 | flag_cinterv_3;
DEFINE flag_inhib_arm := flag_cinterv_2 | flag_cinterv_3;
DEFINE flag_st_1 := base_speed.v = 1 & arm_pos.v = 0;
DEFINE flag_st_4 := base_speed.v = 1 & arm_pos.v = 1;
DEFINE flag_st_5 := base_speed.v = 2 & arm_pos.v = 1;
DEFINE flag_cinterv_1 := flag_st_1 ;
DEFINE flag_cinterv_2 := flag_st_4 | flag_st_5 ;
DEFINE flag_cinterv_3 := FALSE ;
```

The other computed criteria are: number of states where an intervention is applied, use of combined interventions, i.e., application of several interventions on the same state, the type of intervention. For example, our strategy makes use of the two defined interventions `brake_base` and `inhib_arm` and uses no combination of interventions. This strategy is the same as in Figure 4.

Our modeling and synthesis methods find the same strategy that was previously found intuitively on the graphical representation. Interventions are clearly modeled, contrary to the graphical method. Moreover, as our modeling is textual we can solve the same problem type with three or more variables.

4.2 Boxes Sliding from the Base

The robot arm has an impactive gripper as an end-effector that takes and places boxes on its base, which can be used to convey part boxes. In this case, the robot must respect a speed limit V_1 that is less than the general speed limit.

Discrete Model. The available observations are: 1) *box*, a Boolean (*true* in presence of box), and 2) base speed v (the same as in Section 4.1). The safety invariant is $box = false \vee v \leq V_1$. A safety margin value can be placed on speed. The resulting integer ranges are ⟦0,1⟧ for *box* and ⟦0,2⟧ for speed (with `cata:= box=1 & speed=2`). The only possible intervention is braking, since the presence of boxes is not controllable.

Synthesis. Running the automatic method returns no strategy. During interactive exploration, braking is suggested and applied, leading to a complete loss of permissiveness in the state `box=0 & speed=2`. In other words, the robot cannot go faster than V_1 even if there is no box on the base. This is clearly not acceptable. The user can choose either another suggested intervention (not possible in this example) or an earlier state in the path (which brings here no benefit). The current model and requirements admit no satisfying strategy. The intuitive cause is that the presence of a box is uncontrollable.

Now, according to the robot service hypotheses only the robot arm is allowed to place a box on the base. We add to the model the observable variable `gripper` with values {`closed_empty`, `open`, `closed_with_box`} and the associated interventions: `inhibit_opening` and `inhibit_closing`. The variable is continuous in the sense that from the value `closed_empty` to `closed_with_box`, the gripper always transits by `open`. We model that a box cannot arrive on the base

without being in the gripper, and symmetrically a box can only be removed by the (open) gripper.[2]

```
TRANS box.v=0 & next(box.v)=1 -> gripper.v=closed_with_box
TRANS box.v=1 & next(box.v)=0 -> gripper.v=open
```

Due to perception latency, gripper and box sensors may not be synchronized. Therefore, the likely next `gripper` values (`open` in the first constraint) are not specified.

The automatic method returns 32 strategies (which is a lot, so the selectivity of the method should be improved). For every strategy, 5 states are not continuously reachable and braking is applied in every warning state. One strategy uses only braking. The other strategies add some inhibitions to this minimal strategy. For instance, we consider the strategy that adds `inhibit_opening` in warning states with no box on the base (`inhibit_closing` is not used).

In this example, the safety invariant is first ensured, with a high impact on permissiveness. The gripper hypothesis makes the safety invariant feasible without any impact on permissiveness at the expense of lower safety coverage. Even if modeled safety is fully checked, the strategy does not cover all cases, e.g., when workers disobey service regulations and place boxes on the robot base.

4.3 Consistency between Strategies

The two models with their strategies make every intervention pair reachable. By modeling that $V_0 < V_1 - m$, the braking triggered by the first strategy is no longer concurrent with interventions of the second invariant. In our example, `inhib_arm` is compatible with `brake` and `inhibit_opening`.

5 Related Work

Several safety monitoring approaches have been proposed in the literature. For instance [6] argues for a small and simple component in charge of guaranteeing system safety, in particular with respect to hazardous sequences of function invocations. We actually extend this conceptual approach proposing a systematic methodology for the identification of the properties to ensure, while focusing only on invariants.

Runtime verification (RV) typically generates code instrumentation from temporal logic properties to verify execution traces at runtime [7]. Runtime verification can be seen as a downstream process of our workflow: it could implement the monitor from the specification that we generate. Some runtime verification work explores the issue of independence between the monitor and the monitored system. For example, Pike *et al.* [8] consider time-triggered monitoring of a set of global variables, which avoids code instrumentation, achieves time-isolation, and consequently does not require re-certification of the system due to the presence of the monitor. A concept close to permissiveness is defined as functionality: *"the monitor cannot change the monitored system's behavior, unless the latter*

[2] For clarity, the gripper variable is given textual values rather than integer values.

has violated its specification." Another relation between RV and our work is the use of formal verification for monitoring purposes. We check offline the tree of all possible executions (of the model) by using the branching logic CTL whereas RV checks concrete executed traces with respect to linear temporal properties. The reaction to trigger when detecting an error is called the steering problem in the runtime verification community. It is a potential feature of monitors, but it remains much less developed than the detection part. Error detection typically returns information to the monitored program or raises an exception. Other possible reactions are considered as ad-hoc to particular systems because they are not formally captured.

A parallel can be established between game theory and the way the system is modeled, as possible physical behaviors. The monitor player is able to fire or inhibit some transitions whereas the opponent, which can be regarded as the environment or the main controller, is able to fire any transitions. Safety rules are then the monitor strategy to achieve the winning condition (safety, permissiveness and validity) whatever the opponent plays. In particular, we take inspiration from *supervisor synthesis* [9], which is close to game theory.

Supervisor synthesis is based on language theory. It outputs directly the maximally permissive monitor, i.e., the monitor resulting in the system automaton that recognizes the largest language. Therefore, permissiveness is taken into account but the user cannot customize it, by preferring one state instead of another. In [10], Fotoohi *et al.* use supervisor synthesis to check the safety requirements of a semi-autonomous wheel-chair.

Woodman *et al.* [11] present a very similar workflow to monitor autonomous systems. They use HAZOP to identify hazards and determine (intuitively) the corresponding safety rules, which are if-then-else rules. From sensor observations, the monitor (safety layer) sends actuation inhibitions to both the controller and the software actuator interface. The strong point of the method is to take into account sensor uncertainty. Permissiveness is implicit.

6 Conclusion

We have described a method for obtaining a high-level safety monitor specification, taking into account the specific features of autonomous systems. We base it on hazard analysis, which is non-formal. Thanks to formal methods, we ensure that the derivation from formal safety invariants to safety rules is correct, provided the modeling of safety invariants is valid. Safety invariants are modeled separately in order to maintain model validability and to ensure scalability.

Our method justifies the modeling effort in that it does not only check the specification but also guides the user in building it. Compared with related work, both actions and inhibitions are allowed, resulting in a more generic method. Another strong point is the explicit modeling of permissiveness. The user has no permissiveness requirement to provide and can choose precisely the permissiveness trade-off (provided variable dependency is modeled). By using the template, the modeling approach is scalable to many variables and interventions.

As future work, the algorithm selectivity is to be improved and the method extended to process safety invariants other than those based on thresholds. The method has yet to be applied on real and complete systems. Implementation of the monitor would show how to adapt our hypotheses to the real system, or vice versa. The implemented safety interventions have to comply with the temporal hypothesis of the method taking into account the system dynamics and the environment: 1) inhibitions have to be effective "instantaneously"; 2) margin values have to cater for possible action latency. Note that the permissiveness analysis always prefers inhibitions to actions whereas actions may be preferred from an implementation viewpoint. Future work concerns customization of the fault independence assumption by implementing safety rules at different levels in the system architecture, resulting in several safety monitors instead of one.

Acknowlegment. This work is partially supported by the SAPHARI Project, funded under the 7th Framework Programme of the European Community.

References

1. Mekki-Mokhtar, A., Blanquart, J.P., Guiochet, J., Powell, D., Roy, M.: Safety trigger conditions for critical autonomous systems. In: 18th Pacific Rim Int'l Symp. on Dependable Computing (PRDC), pp. 61–69. IEEE (2012)
2. ISO/IEC 61508-7: Functional safety of electrical / electronic / programmable electronic safety-related systems - part 7: Overview of techniques and measures (2010)
3. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
4. Dufossé, F., Machin, M., Guiochet, J., Powell, D., Roy, M., Waeselynck, H.: Safety strategy synthesis: Game theory versus model-checking. LAAS-CNRS, Tech. Rep. 14059 (2014)
5. Saphari project, `http://www.saphari.eu`
6. Rushby, J.: Kernels for safety. Safe and Secure Computing Systems, 210–220 (1989)
7. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
8. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: 2nd Int'l Conf. on Runtime Verification, San Francisco, California, USA (2011)
9. Wonham, W.M.: Supervisory control of discrete event systems (2005)
10. Fotoohi, L., Gräser, A.: A supervisory control approach for safe behavior of service robot case study: Friend. In: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 1305–1306. ACM (2010)
11. Woodman, R., Winfield, A.F., Harper, C., Fraser, M.: Building safer robots: Safety driven control. Int'l J. Robotics Research 31(13), 1603–1626 (2012)