

Model-Checking and Game theory for Synthesis of Safety Rules

Mathilde Machin, Fanny Dufossé, Jérémie Guiochet, David Powell, Matthieu Roy, Hélène Waeselynck

▶ **To cite this version:**

Mathilde Machin, Fanny Dufossé, Jérémie Guiochet, David Powell, Matthieu Roy, et al.. Model-Checking and Game theory for Synthesis of Safety Rules. 2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE), Jan 2015, Daytona Beach Shores, United States. pp.36-43, 10.1109/HASE.2015.15 . hal-01164981

HAL Id: hal-01164981
https://hal.science/hal-01164981

Submitted on 18 Jun 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model-checking and Game Theory for the Synthesis of Safety Rules

Mathilde Machin, Fanny Dufossé, Jérémie Guiochet, David Powell, Matthieu Roy, Hélène Waeselynck
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Université de Toulouse, LAAS, F-31400 Toulouse, France

***Abstract*—Ensuring that safety requirements are respected is a critical issue for the deployment of hazardous and complex reactive systems. We consider a separate safety channel, called a monitor, that is able to partially observe the system and to trigger safety-ensuring actuations. We address the issue of correctly specifying such a monitor with respect to safety and liveness requirements. Two safety requirement synthesis programs are presented and compared. Based on a formal model of the system and its hazards, they compute a monitor behavior that ensures system safety without unduly compromising system liveness. The first program uses the model-checker NuSMV to check safety requirements. These requirements are automatically generated by a branch-and-bound algorithm. Based on a game theory approach, the second program uses the TIGA extension of UPPAAL to synthesize safety requirements, starting from an appropriately reformulated representation of the problem.**

***Index Terms*—Model checking, Game theory, Supervisor synthesis, Safety Monitoring, Safety Rules, Reactive Systems.**

I. Introduction

Ensuring safety is a critical issue for the deployment of hazardous and complex reactive systems such as autonomous robots or uninhabited vehicles. The complexity of such systems makes it difficult to check their correctness statically. Therefore, we consider the implementation of runtime safety measures in a device called a *safety monitor* [1] that is independent from the main controller channel and simple enough to be verifiable. The monitor is equipped with the necessary means for context observation (i.e., sensors) and able to trigger, when necessary, appropriate safety interventions. The monitor is in charge of ensuring a safe behavior of the system with minimal impact on the system's functionality, which we characterize by the notion of *permissiveness*. The monitor behavior is specified declaratively by a set of *safety rules* of the form: if *condition* then *intervention*.

We address the issue of specifying such a monitor. Our previous work [2] defined a process to elicit the safety rules starting from an analysis of hazards. This is an offline process that aims to produce safety rules that can be implemented in an online safety monitor (implementation issues are however not discussed in this paper). While the process was originally manual, we recently started to investigate the use of formal methods to automate some of its steps [3]. This paper focuses on one of the steps: the synthesis of permissive rules for a given safety invariant. It presents two alternative rule synthesis programs we developed based on off-the-shelf tools, respectively a model checker and a game theory tool.

First, Section II presents the concepts and the formal support for the safety rules and Section III the synthesis approach. Then, the two safety rule synthesis programs are presented: in Section IV, a program using the model checker NuSMV [4] is developed to provide all satisfying safety rules; in Section V, a game theory model allows the rules to be synthesized by TIGA [5], the game theory extension of the UPPAAL model-checker. These rule synthesis tools are compared in Section VI. Section VII discusses related work.

II. Formalisation

We first present the safety monitoring concepts and definitions to formally express a discrete model of the system, and the properties the safety rules should comply with.

A. Concepts

We consider safety monitors as described in the IEC 61508 standard [1]. We assume the following:

- The monitor is independent of the control channel.
- Variable values observed by the monitor are correct.
- Monitor reaction is faster than the control channel.
- Safety interventions are always successful.

The monitor must accommodate any variation of the environment and any dysfunction of the control channel, including arbitrary behavior of the latter, e.g., when it is faulty.

We propose a method based on a hazard analysis to synthesize safety rules that together specify a safety monitor. This hazard analysis results in a set of safety invariants. A **safety invariant (SI)** is a property that ensures hazard avoidance. SI violation places the system in a catastrophic state, from which we assume that no recovery is possible. To prevent SI violation, the monitor is able to trigger safety interventions. Safety interventions are expressed as constraints on how system variables can change. Safety rule synthesis consists in deciding what intervention to apply and when. We distinguish two types of **safety interventions**:

- A **safety inhibition** prevents a change in system state. When triggered, an inhibition is supposed to be immediately operational.
- A **safety action** triggers a change in system state (and implicitly prevents other state changes).

A **safety rule** is a high-level specification of the monitor, of the form: *if* [safety trigger condition] *then* [safety intervention]. The intervention is applied when the safety

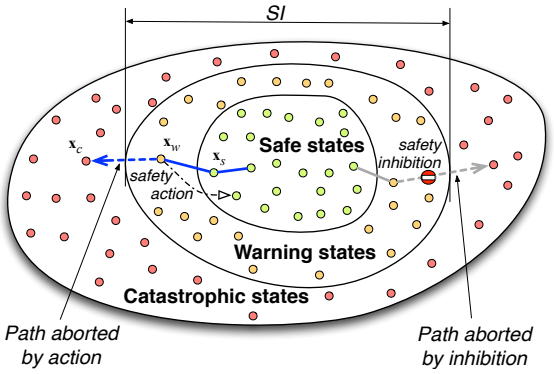


Fig. 1: Partition in catastrophic, warning and safe states

trigger condition is true. This condition has also to meet the intervention precondition. For example, locking a door is only meaningful when the door is closed.

As illustrated in Fig. 1, the safety invariant (SI) defines a partition between catastrophic states and non-catastrophic states of the system. Interventions have to be applied to prevent catastrophes. They thus add constraints to the physically possible system behavior to prevent SI violation. The set of non-catastrophic states is partitioned into warning states, where interventions are required, and safe states, in which the system operates without constraint. The warning states, which correspond to the safety margin, are defined such that every path from a safe state x_s to a catastrophic state x_c, passes through a warning state x_w (see Fig. 1).

The objective of the method is to provide for each SI a set of safety rules to avoid the violation of SI. We define this set of safety rules as a **satisfying safety strategy**, that ensures:

- **Safety**, the ability to ensure that the safety invariant is never violated, i.e., that catastrophic states are unreachable.
- **Permissiveness**, the ability to allow the system to perform its tasks.
- **Validity**, that ensures that no intervention is applied while its precondition does not hold.

Safety and permissiveness are antagonistic. We take this antagonism into account by designing the strategy to be *maximally permissive with respect to safety*, i.e., to restrict the permissiveness only to the extent necessary to ensure safety. However, since interventions must be applied in non-catastrophic states, some reduction in permissiveness must usually be accepted in order to ensure safety. The more the safety margin is extended, the less the monitor is permissive.

B. Discrete model

A safety invariant is expressed formally in the discrete model 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 most amenable to formal verification and is used in many real systems. Moreover, to keep models simple enough to be validated, we model each invariant in a separate model.

We consider as a running example a mobile robot with a folding manipulator arm, with the two following monitor observations: v the velocity of robot movement, in absolute value, and $folded_{arm}$ a Boolean observation of the arm position. The considered safety invariant is $v < V_0 \vee folded_{arm} = true$, i.e., the robot must either respect a (low) velocity limit V_0 or keep its arm folded. The safety monitor interventions are: braking (further denoted by b) and locking the arm (denoted by (a, as precondition, the arm must already be folded). Interventions are modeled by use of the observable variables.

To specify the set of warning states, we try to partition the value sets defined by the safety invariant for each variable. For example, the velocity interval $[0, V_0[$ from the safety invariant is partitionable, according to a safety margin m, into two intervals $[0, V_0 - m[$ and $[V_0 - m, V_0[$. This margin will enable interventions to be applied only in the states where $v \in [V_0 - m, V_0[$. As the system cannot reach $v \geq V_0$ (set of catastrophic states) without passing through $[V_0 - m, V_0[$ (set of warning states), an adequate intervention applied in $[V_0 - m, V_0[$ should prevent the system from reaching states where $v \geq V_0$ (see Fig. 1). In the case of arm position, the observation is Boolean. The set of values is the singleton $\{true\}$, so it cannot be partitioned and no margin exists.

The domains of one observable variable is partitioned, and each class of partition is now considered as a enumerated value. For example, $v \in \{[0, V_0 - m[$ may be represented by `v=0`, $v \in [V_0 - m, V_0[$ by `v=1`. The discrete model is the Cartesian product of the classes of the partitions. For example, Fig. 2 is the model built from the domains of v and $folded_{arm}$. Each state of the discrete model is a *region state*, i.e., a subset of system states. As defined, the partitions ensure that each region contains only catastrophic system states or only non-catastrophic system states. One region state is defined as initial. The warning region states are those that lead the system to the catastrophic region state in one step.

No assumption is made about the controller behavior. The system model contains what is physically possible in the system without a monitor. Safety interventions only remove transitions. , i.e. possible behaviors, and cannot add transitions, i.e., add physically impossible behaviors. Interventions model

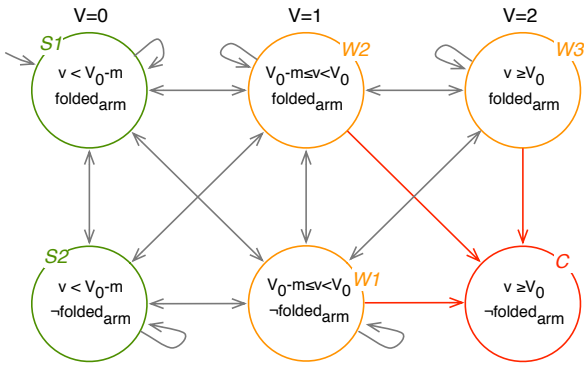


Fig. 2: The example discrete model with partitions $\{true, false\}$ for arm folding, and $\{[0, V_0 - m[, [V_0 - m, V_0[, [V_0, V_{max}[\}$ for velocity

the fact that some transitions can be fired or inhibited by the monitor. We assume that the monitor is always faster than the system, i.e., when the system fires a transition, the monitor always has enough time to apply interventions before the system fires another transition. This assumption mainly relies on the margin values, which must be calculated according to the system dynamics.

The discrete model is formally defined by a set of variables and their partitioned domains, the dependencies on these variables, the definition of the catastrophic states, and the declaration of the possible interventions. The graph produced in Fig. 2 presents the region state model of the running example. Such a visual representation is only presented for explanatory purposes. We use a textual representation of the graph, which is more amenable to automated analysis.

C. Formal safety, permissiveness and validity properties

Applying an intervention consists in removing some transitions from the discrete model. This is of course necessary to ensure safety. Nevertheless, some removed transitions may prevent the system from normally operating, which is a permissiveness problem.

To assess the strategies that will be added to the previous model, we need a formal definition of the safety, permissiveness and validity properties. To that end, we use CTL (*Computation Tree Logic*). 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 a branching operator and a time operator. It is applied to states, or more generally, to statements about the system state.

Safety of the system is naturally modeled as a temporal property. Let $cata$ be the negation of the safety invariant. A strategy N is safe if N satisfies $AG \neg\, cata$.

Permissiveness is translated by three liveness properties. In the CTL formulation, these properties are applied to each non-catastrophic state. Let V_{nc} be the set of non-catastrophic states.

- **Simple liveness**: $\forall s_{nc} \in V_{nc}, EF\ s_{nc}$.
  Any non-catastrophic state is reachable from the initial state.
- **Universal liveness**: $\forall s_{nc} \in V_{nc}, AG\ EF\ s_{nc}$.
  Any non-catastrophic state is reachable from any reachable state.
- **Continuous (and universal) liveness**:
  $\forall s_{nc} \in V_{nc}, AG\ EF\ \big(s_{nc} \wedge EG\ s_{nc}\big)$.
  Any non-catastrophic 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.

We choose to specify permissiveness according to these three ordered properties, because permissiveness can be graded. Indeed, it is usually impossible to ensure safety without some loss of permissiveness, particularly with respect to continuous liveness.

For safety, we pessimistically consider that several variables may change their values simultaneously and independently. We call such simultaneous modifications *diagonal transitions* by reference to the two variable case (see Fig. 2). From the permissiveness point of view, relying on those possible but unlikely transitions to ensure liveness is not desirable. A more complete definition of liveness properties that ignore diagonal transitions during permissiveness checking is provided in [3] and used by the tools we developed.

Validity checks that interventions are not applied in states that violate their preconditions. Such an invariant is defined as:

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

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

III. Synthesis principle

In this paper, we consider that safety invariants (SI) have been identified through hazard analysis. Then, the SI are analyzed one by one. For each SI, the objective is to identify a safe and permissive strategy. As presented before, a strategy is a set of safety rules. Each safety rule specifies the application of an intervention in a warning state. Let I be the set of interventions, of size m, and $I_C = 2^I$ the set of *intervention combinations*. For a given SI, if we consider n warning states, a strategy will be noted $N = (i_1, \ldots, i_j, \ldots, i_n)$, where $i_j \in I_C$ is the application of an intervention combination (possibly a single intervention) to the warning state j. For example, in a model with two possible interventions a and b, and three warning states, the strategy denoted $N_1 = (\{a\}, \{a, b\}, \emptyset)$ means: intervention a is applied in state 0, both a and b in state 1 and no intervention in state 2.

For the general case, with n warning states, and m interventions, the number of possible strategies is 2^{mn}. Among these strategies, the objective is to determine *satisfying* strategies, i.e., strategies that are valid, safe and permissive.

Using existing verification tools, we consider two approaches for discovering satisfying strategies. The first one, presented in Section IV, synthesizes strategies through an exploration algorithm, and then uses a model checker (NuSMV) to verify safety and permissiveness of these strategies. The second approach, presented in section V, uses a tool for game theory (TIGA) to synthesize winning strategies.

As safety invariants are analyzed separately, strategies might be conflicting. Checking their consistency is a downstream step, which is not addressed in this paper.

IV. Strategy synthesis using NuSMV

A basic synthesis algorithm could enumerate every possible strategy to check whether it satisfies the requested properties. But complete enumeration is not desirable due to the exponential number of strategies. We designed an optimized method to explore and prune the tree of strategies (branch-and-bound algorithm). Notably, we focus on *minimal* satisfying strategies, i.e., strategies for which each intervention is necessary. A satisfying strategy $N = (i_1, \ldots, i_n)$ is minimal if there does

not exist a different satisfying strategy $N' = (i'_1, \ldots, i'_n)$ such that $i'_1 \subseteq i_1, \ldots, i'_n \subseteq i_n$. During exploration, the branch-and-bound algorithm calls NuSMV to check the pruning conditions. The choice of NuSMV is discussed in Section VII.

A. Discrete model and properties in NuSMV

The formalization of our problem in NuSMV is natural and consists of a straightforward rewriting of properties defined in Section II-C: validity (`valid`), safety (`safe`) and permissiveness (`perm`) with a focus on universal liveness.

SMV enables the declaration of integer variables and of constraints on their behavior. The dynamic aspect is expressed using the `next` operator. From variable ranges, NuSMV builds transparently the Cartesian product. When no constraint is declared, all combinations of variable values (i.e., region states) are possible and all transitions between each pair of states are implicitly declared. Then constraints are added to delete states and transitions that have no physical meaning or are not physically feasible due to dependencies between variables. In our discrete model, the continuity constraint for variables is expressed as `next(x) :={x,x+1,x-1}` (which means that next value of x will be x, $x+1$ or $x-1$). For example, the braking action is modeled by:

$$braking \rightarrow ((v \neq 0 \rightarrow next(v) = v - 1) \wedge (v = 0 \rightarrow next(v) = 0))$$

B. Branch-and-bound algorithm for strategy synthesis

NuSMV is first used to automatically extract the warning states. We can thus limit the search to strategies with interventions applied in warning states only, rather than in all non-catastrophic states. Then, we use a branch-and-bound algorithm to explore candidate strategies. Fig. 3 presents the resulting search tree for our running example with three warning states. In the figure, we adopt a simplified notation where strategy $(\{a\}, \{a, b\}, \emptyset)$ is noted $(a, a.b, 0)$. An undefined intervention is denoted -1. Exploration starts with the undefined strategy $(-1, -1, -1)$ and examines each warning state to decide the interventions to apply in it. The children

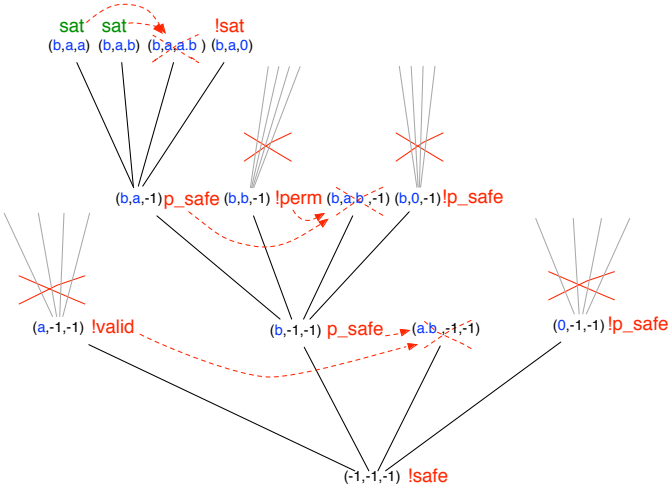


Fig. 3: Search tree for our example

of a partially defined strategy $N = (i_1, \ldots, i_p, -1, \ldots, -1)$ are potentially the 2^m nodes $(i_1, \ldots, i_p, i_{p+1}, -1, \ldots, -1)$, with $i_{p+1} \in I_C$, and m the number of interventions. However, execution does not traverse the complete tree. Pruning criteria are used to identify subtrees that can be discarded from the search without losing any minimal satisfying strategies. As shown visually in Fig. 3, these criteria may allow us to ignore the descendants of the current node under consideration as well as some of its sibling nodes. For example, neither the descendants of $(a, -1, -1)$ nor its sibling node $(a.b, -1, -1)$ are explored.

Table I gives an overview of the pruning criteria. They are assessed using calls to NuSMV. Given the partial strategy N under consideration, our tool automatically produces the model and properties to check. In particular, the original model is automatically changed to insert the trigger conditions of interventions corresponding to N, with -1 interpreted as no intervention. For each criterion, the discarded strategies either don't satisfy one of the required properties of validity, permissiveness and safety, or are not minimal.

The first two criteria discard strategies that are not valid (`!valid`) or not permissive (`!perm`).

Assume partial strategy N is `!valid`. For instance, in Fig. 3, strategy $N = (a, -1, -1)$ is `!valid` because the intervention that locks the arm folded is applied in a warning state where the arm is unfolded. Child strategies may define interventions in other warning states, but this will not fix the problem in the first one. Either the first warning state becomes unreachable, and so the child strategy is not permissive, or the state is reachable and the intervention's precondition is violated. So, the children of an invalid strategy are either invalid or not permissive. Consider now a partial strategy that is not permissive. All its children are `!perm` as well, because adding interventions can only remove transitions. So, if a partial strategy is either `!valid` or `!perm`, none of its children – and recursively none of its descendants – can be a satisfying strategy. We can cut the subtree without losing solutions.

Now, consider the siblings of N. Of particular interest are the siblings that apply a superset of N's interventions in the currently decided warning state. Formally, if $N = (i_1, \ldots, i_p, -1, \ldots, -1)$, let us consider the siblings $N' = (i_1, \ldots, i'_p, -1, \ldots, -1)$ with $i_p \subset i'_p$. In Table I, such siblings are called *combined siblings*. They differ from N only in the p^{th} warning state, where additional interventions are combined with N's ones. For example, $(a.b, -1, -1)$ is a combined sibling of $(a, -1, -1)$. It is trivial that if N is `!valid`, so is N', and similarly for `!perm`. So, the first two criteria allow us to cut not only the descendants of N, but also the subtrees of all combined siblings of N.

The third criterion discards strategies that are not minimal. Assume N is a satisfying strategy (i.e., safe, valid and permissive), which we note `sat`. Its descendants and combined siblings might be `sat` as well, but they involve additional interventions and so are not minimal. The corresponding subtrees can be pruned. N is appended to the list of solutions

TABLE I: Pruning criteria

	Node property	Pruned relative nodes
1	`!valid`	Descendants and combined siblings
2	`!perm`	Descendants and combined siblings
3	`sat`	Descendants and combined siblings
4	`!p_safe`	Descendants
5	`p_safe`	Combined siblings

returned by the search.

The fourth and fifth criteria are evaluated using a *submodel* where the warning states with an undefined (-1) intervention are removed. This submodel focuses on reaching the catastrophic state via the warning states for which a decision has been taken, and specifically via the state s_p targeted by the most recent decision. If the strategy is `safe` in the submodel, we say that it is partially safe (`p_safe`).

Assume N is `!p_safe`. There is a path to the catastrophic state in the submodel. The descendants of N cannot remove this path. Their added interventions can only delete transitions exiting warning states that are outside the current submodel. So, all descendants of N are unsafe and can be pruned (fourth pruning criterion).

The last criterion concerns the opposite case where N is `p_safe`. It prunes non-minimal sibling strategies that apply a set of interventions i'_p where $i_p \subset i'_p$ is sufficient. This criterion is the most difficult to explain. Its detailed justification depends on the reachability of s_p in the submodel, which can be controlled by the order in which the warning states are processed. In the algorithm we implemented, an adequate order is determined on-the-fly, allowing for efficient pruning.

To conclude, we have five criteria that discard strategies that are either invalid, not permissive, unsafe or not minimal. Moreover, we have a demonstration (not presented in this paper) that their joint use discards all the non-minimal strategies, i.e., the search returns exactly the set of minimal satisfying strategies.

C. Running example

Fig. 3 presents the resulting tree for the example introduced in Fig. 2, where braking is labeled by b (reduce the velocity), inhibition of the arm movement by a, and their combination $a.b$. To simplify, we consider a static adequate ordering of the warning states: $W1$, $W2$, $W3$. The tree traversal is then executed as follows. The root empty strategy is first examined and assessed as `!safe`. Then, its first child, with strategy $(a, -1, -1)$ (corresponding to arm inhibition in $W1$), is considered. Given that the arm is unfolded in $W1$, inhibition is `!valid`. Then, we cut the corresponding subtree and keep in memory that any combined sibling $(a.x, -1, -1)$ should not be considered.

The node labeled by $(b, -1, -1)$ is then assessed. The corresponding submodel for `p_safe` assessment contains the safe states, the catastrophic state C and the warning state $W1$, in which braking intervention is applied. Thanks to this intervention, C is unreachable in the submodel and the strategy is `p_safe`. We cut the combined siblings. Then we examine the first child node $(b, a, -1)$, that is also `p_safe`. We thus cut its combined siblings and consider its children.

The first child (b, a, a) is a `sat` strategy. We thus add it to the list of solutions and do not consider the sibling $(b, a, a.b)$ that is not minimal. (b, a, b) is also `sat` and kept as a solution. $(b, a, 0)$ is `!sat`. The next node in prefix order is labeled by $(b, b, -1)$, corresponding to braking in $W1$ and $W2$. This strategy forbids any path to $W3$ and is thus `!perm`. We cut the subtree and the combined siblings.

The next partial strategy is $(b, 0, -1)$. The submodel for `p_safe` property contains all nodes except $W3$ and has no intervention in $W2$, which permits a direct transition to catastrophic state C. The strategy is thus `!p_safe`. It ends the exploration of the subtree rooted by $(b, -1, -1)$.

Then, as $(a, -1, -1)$ was `!valid` and $(b, -1, -1)$ was `p_safe`, $(a.b, -1, -1)$ is not considered. The submodel of the last node $(0, -1, -1)$ corresponds to a submodel with the states $S1$, $S2$, $W1$ and C without any intervention in $W1$, permitting a direct transition to C. It is thus not `p_safe` and its subtree is cut. The tree traversal is terminated, leading to two minimal computed strategies: (1) (b, a, a), which specifies that the brake should be engaged when velocity is in the margin and arm unfolded ($W1$), and that the arm should be blocked in the folded position when arm is folded and velocity is in the margin ($W2$) or over the limit ($W3$); and (2) (b, a, b), which specifies similar rules except that in $W3$ it brakes instead of inhibiting the arm.

Over the 85 nodes of the tree, that correspond to $2^6 = 64$ possible strategies, only 9 strategies have been explored to compute these 2 minimal satisfying strategies.

V. Strategy synthesis using UPPAAL-TIGA

The second approach we studied uses game theory. Intuitively, it considers a 2-player game where the safety monitor plays against a malicious adversary that stands for a faulty controller or a disobliging environment. The adversary fires transitions in the system model. Its aim is to reach a catastrophic state. To counteract the adversary's moves, the monitor may trigger safety interventions. It has a winning strategy if it always succeeds in maintaining both safety and universal liveness, whatever the moves of the adversary.

We thus need to encode the safety strategy synthesis problem as a game solving problem. A difficulty is that the solving algorithms typically consider either safety properties AGP or reachability ones AFP. In our case, we would like to consider both safety and universal liveness, yielding reachability properties of the form $AG\ EF\ P$. We are not aware of any off-the-shelf game tool allowing us to synthesize winning strategies for such properties. We thus had to bypass the difficulty by adopting a stronger property (universal liveness in a *bounded* number of steps) and by introducing modeling artifacts to properly encode the synthesis problem.

A. TIGA Modeling

The game theory tool used in this paper is the UPPAAL extension for game theory, TIGA, as developed in Section VII.

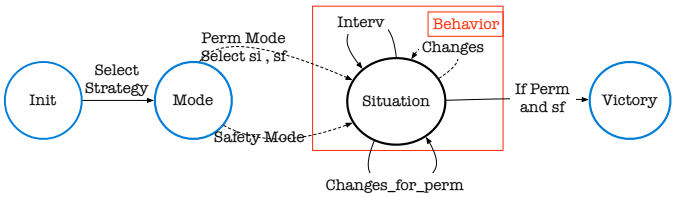


Fig. 4: Schematic view of the encoding as a game

In UPPAAL, a transition system is represented by a network of concurrent automata. An interleaving semantics is adopted, that is, concurrent transitions cannot be simultaneously fired unless they are explicitly synchronized. The modeling paradigm is thus different from the one of NuSMV, where states are implicitly defined via sets of variable values and an arbitrary number of variables may simultaneously change their value in one step (what we called a *diagonal transition*). The difference impacts the way we model the system behavior. In order to account for diagonal transitions, we do not consider separate automata for the various variables, but rather a single automaton with macro-transitions that assign values to (subsets of) variables.

Fig. 4 gives a schematic view of how we encode the synthesis problem as a game solving problem. Its core part is the system `Behavior` model where:

- `Situation` is a macro-state capturing all possible values for the system variables (e.g., values for velocity and folded arm in the running example).
- `Changes` is a macro-transition capturing all possible changes of variable values (e.g., the velocity is increased by one, or both the velocity and arm position change).
- `Interv` is a macro-transition capturing the application of any subset of interventions (e.g., the application of a braking action or an inhibition of arm unfolding).

`Changes` is controlled by the malicious adversary (which is indicated by a dotted line in Fig. 4) while `Interv` is controlled by the monitor (indicated by a solid line). The model is untimed, except for a clock that forces the players to move: a state invariant bounds the time that can be spent in `Situation` without firing any transition. The turns of the players are controlled by an auxiliary Boolean parameter. After the adversary has fired a change in the system variables, it always hands the dice on to the monitor, which decides whether or not to apply interventions, according to its strategy. Note that the monitor cannot decide just to block the adversary without firing any intervention, because this would violate the state invariant in `Situation`.

All extra states and transitions that appear in Fig. 4 outside of the `Behavior` part, are modeling artifacts to encode the synthesis problem. Since universal liveness is not expressible in TIGA, we considered a stronger property: any non-catastrophic state is reachable from any other non-catastrophic state in less than k steps.

This property corresponds to a game where the adversary chooses an arbitrary pair of non catastrophic states s_i and s_f and the monitor is challenged to find a path of length less than k between them. This game is quite different from the classical safety game, in which the adversary would try to reach the catastrophic state. In the permissiveness game, the monitor controls the system transitions to demonstrate the existence of a path. Both the safety and permissiveness games must be encoded in the same formal model, because both properties are required from any monitor's strategy. The consideration for the two games is explicit in the left part of Fig. 4: the monitor first chooses the strategy, and then the adversary chooses the game. The order of the choices is very important. If we did not have an initial step that explicitly establishes the strategy once and for all games, the monitor could make its strategy dependent on the game. In the case of a safety game, it would apply as many interventions as it can to avoid the catastrophic state, while in the permissiveness game it would apply none. Rather, in the model of Fig. 4, the monitor must decide its strategy without knowing which game will be chosen by the adversary. The monitor's strategy should be winning irrespective of the chosen game, and in case the permissiveness game is chosen, irrespective of the pair s_i, s_f. Auxiliary control variables are added to store the chosen strategy and games, where in particular `perm` is assigned true for a permissiveness game and false for a safety game.

Since the strategy is decided from the very beginning, we must ensure that the monitor's moves comply with it. The `Interv` macro-transition is modeled in such a way that the triggering of interventions is governed by the value of the stored strategy. In the permissiveness game, the monitor does not only apply interventions but also fires the other system transitions. To allow this, we had to duplicate the `Changes` macro-transition into a new macro-transition, `Changes_for_perm` that is controllable by the monitor (note the solid line). `Changes` and `Changes_for_perm` are guarded by the `perm` parameter, indicating the type of the current game. A safety game enables `Changes` and disables `Changes_for_perm`, and reciprocally in a permissiveness game. Note that contrarily to `Changes`, `Changes_for_perm` does not allow for multiple variable assignments (e.g., either the velocity changes in one step, or the arm position, but not both). In our liveness properties, reachability should not depend on diagonal transitions.

Finally, we specify the global winning condition that captures both safety and permissiveness. The monitor wins iff:

$$AG\,(\neg\, cata \wedge perm \rightarrow steps < k)$$

where `steps` is an instrumentation variable that counts the number of transitions fired in `Situation`. During a permissiveness game, reaching the target state s_f activates a transition to an auxiliary state `Victory` in which `steps` no longer evolves (see the right part of Fig. 4). Since the `Situation` invariant forces the players to keep moving, `steps` inexorably increases unless s_f is reached and `Situation` is left for `Victory`. Hence, formula $perm \rightarrow steps < k$ expresses the intended property: in case the adversary choses a permissiveness game, the monitor must find a bounded path between any arbitrary s_i and s_f.

B. Running example

Let us consider the example of Fig. 2 with three strategies (b,b,0), (b,0,0) and (b,a,a) with the notation of Fig. 3:

(b,b,0): The strategy is to brake in $W1$ and $W2$, and no intervention in $W3$. In the Safety Mode, the game starts in state $s1$. From there it is not able to reach a catastrophic state: the strategy is safe. In the Permissiveness Mode, with nodes $s_i = s1$ and $s_f = W3$, the safety strategy prohibits velocity greater than V_0, thus s_f is not reachable. The strategy is not permissive. Hence, the winning condition is not satisfied.

(b,0,0): The selected strategy is to brake in $W1$ and no intervention in $W2$ and $W3$. In the Safety Mode, the game reaches vertex `Situation` in initial state $s1$. Then, the opponent can trigger transitions to $W2$, $W3$ and C without any safety intervention. The strategy is thus unsafe.

(b,a,a): The selected strategy is to brake in $W1$ and inhibit the arm in $W2$ and $W3$. In the Permissiveness Mode, the monitor can reach s_f from s_i for every pair of non-catastrophic states s_i and s_f, and in the Safety Mode, the opponent cannot reach state C. This safety strategy is thus safe and permissive. The winning condition is satisfied.

VI. TIGA and NuSMV: Comparison

The implementation of both tools involved very different issues. With NuSMV, a specific synthesis program has been developed in C/C++, calling the model checker as a function. NuSMV is only a part of the synthesis tool, but is well suited to model the system and properties. The main issue was the identification of the pruning criteria for our algorithm.

On the contrary, TIGA's existing algorithm can carry out synthesis completely and the major issue is the modeling. Formalizing the synthesis problem as a game problem is done at the expense of modeling artifacts. In particular, the game model has a number of states significantly higher than that of the original system model. In our running example, 2 variables generate 6 region states. The same example in TIGA corresponds to 273 game states due the added artifacts.

The TIGA tool outputs *one* satisfying strategy (the first it finds), whereas the NuSMV tool outputs *all minimal* ones.

An experimental comparison of the tools has been carried out using artificial models. It allows us to consider search spaces ranging for a few thousands strategies to more than 10^{18}. The models are generated as follows. All variables have the same number of values. For example, in Table II, system `2var3val` has two variables and each variable has 3 values. The initial state has all variables at value 0, and the only catastrophic state has variables at their maximum values. There are two possible models for interventions. In the first one, (suffix `_a`), for any variable, the monitor can increase the value of the variable, decrease it, or inhibit any changes of value. In the second case (suffix `_l`), the monitor can only decrease and inhibit the variable values, and in addition, the variables cannot be inhibited when they take their maximum value. For each tool, we identify time and memory costs of the synthesis. Experiments have been done on one core of an Intel Core I7-4770 processor running at 3.4GHz. The results are in Table II.

TABLE II: Experimental compared results

	TIGA Tool		NuSMV Tool	
	Time	Memory	Time	Memory
2var2val_l	0.36s	68.8M	0.08s	11.8M
2var2val_a	2.01s	288M	0.16s	11.9M
2var3val_l	`Out of memory`		1.3s	12.2M
2var3val_a	`Out of memory`		3.5s	12.3M
3var2val_l	`Out of memory`		15s	12.3M
3var2val_a	`Out of memory`		35s	12.5M
3var3val_l	`Out of memory`		1h7m	14M
3var2val_a	`Out of memory`		2h6m	14.3M

TABLE III: Experimental pruning performance (NuSMV tool)

	Number of strategies	Examined nodes		Number of solutions
		Number	%	
2var2val_l	4096	6	0.15%	0
2var2val_a	262144	8	0.003%	0
2var3val_l	4096	74	1.8%	36
2var3val_a	262144	109	0.04%	36
3var2val_l	$4.39 * 10^{12}$	764	<0.001%	108
3var2val_a	$9.22 * 10^{18}$	1 191	<0.001%	108
3var3val_l	$4.39 * 10^{12}$	94 723	<0.001%	48 000
3var3val_a	$9.22 * 10^{18}$	159 145	<0.001%	48 000

An obvious result is the huge and fastly growing memory requirement of the TIGA tool. It runs out of memory for half of the models with two variables and for all models with three variables. This poor performance can be explained by the fact that TIGA is not designed to serve our specific needs. We had to introduce modeling artifacts and could not tune TIGA's internal algorithm to process them efficiently.

In contrast, the memory consumption of the NuSMV tool is much lower and increases slowly. The increase of the execution time is also kept reasonable, if one considers that there are 15 orders of magnitude in size between the smallest and largest search spaces, and that we do not stop the algorithm after a first solution is found. The pruning criteria allow us to limit the number of steps to explore the tree of strategies.

Table III gives more insights into the pruning. The first columns gives the size of the search space, i.e. the number of complete strategies. It would be the number of steps of brute-force search. The second column compares our algorithm to brute-force search. For example, in the first row, our algorithm visits 6 partial or complete strategies (tree nodes), while brute-force search would visit 4096 strategies ($6/4096 = 0.15\%$). The gain is very high for all models. Finally, the third column gives the number of solutions found by the search. These solutions are a subset of the examined strategies and it is interesting to see how many extra nodes are visited to find them. In the largest model (`3var3val_a`), the total number of visited nodes is only 3.3 times the number of solutions. The pruning criteria are very efficient.

It may be surprising that a model with only three variables requires a 2-hours synthesis. One might wonder whether the approach is useful in realistic cases. Firstly, the number of variables is not unrealistic. A safety invariant models only one safety-relevant aspect of a system. In the real system studied by [2], each invariant had no more than two variables. Secondly, the artificial models we used are generic, i.e., they have many interventions and no variable dependencies. It

follows that there are numerous solutions to find, much more than in real cases. In a case study we are currently performing, a safety invariant has 4 variables (1 boolean variable and 3 variables with three values) and 3 interventions are available. Due to a dependency, the synthesis problem has only 4 minimal solutions. They are found in 3s by our NuSMV tool.

VII. Related work

Runtime verification (RV) typically generates code instrumentation from temporal logic properties to verify execution traces at runtime [6]. As in our work, RV uses 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.

In this paper, we use NuSMV [4], which enables variable-oriented modeling of systems with properties given in CTL. Other model checkers, like SPIN [7] and UPPAAL [8], do not allow a simple implementation of permissiveness (liveness) properties: SPIN only admits LTL properties and UPPAAL does not permit associations of operators AG and EF.

Game theory is used for controller synthesis on a game automaton [9]. In game automata, transitions result from the decisions of both players. The controller synthesis consists in restricting the automata nodes and first player decisions to respect some properties and avoid deadlocks. The approach has been applied to robotic functions as complex as motion planning [10]. Supervisor synthesis [11] is a framework that is very close to a 2-player game. Permissiveness requirements are automatically taken into account, with the drawback that they cannot be modified. Actions are not supported. An example of application for robotics safety can be found in [12].

A large number of tools for game theory exist, for different game models. Gambit [13] targets game trees. PESSOA [14] considers systems defined by first order differential equations. In some tools, the game model is consistent with our model, but either no synthesis tool is provided (e.g., MOCHA [15]) or the supervisor definition does not permit to model actions (Supremica [16]). We finally used the model-checker UPPAAL, with its game theory extension TIGA [5], that is able to compute winning strategies. As explained earlier, UPPAAL cannot implement permissiveness, but TIGA is to our knowledge the only publicly available tool that permits strategy synthesis.

VIII. Conclusion

We have provided two methods for strategy synthesis, able to synthesize safe and permissive strategies for an active safety monitor. Both tools have very different features and performance results. We have succeeded in modeling variable-oriented safety invariants and permissiveness in TIGA, which is state-oriented and supports a small part of CTL. Nevertheless, TIGA performance is not sufficient. In contrast, our tool based on NuSMV provides relatively efficient performance at the price of having to develop a branch-and-bound algorithm to explore efficiently the set of possible strategies.

A parallelized version of the NuSMV-based tool is currently under development to further improve performance. We also have started an industrial case study, which will allow us to demonstrate the application of the tool in the framework of the complete safety methodology defined in [3]. Finally, the synthesized strategies apply interventions only according to the current system state and fixed thresholds. Extensions are under study to accommodate more flexible strategies.

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

References

[1] ISO/IEC 61508-7, "Functional safety of electrical / electronic / programmable electronic safety-related systems - part 7: Overview of techniques and measures," p. 153, 2010.

[2] A. Mekki-Mokhtar, J.-P. Blanquart, J. Guiochet, D. Powell, and M. Roy, "Safety trigger conditions for critical autonomous systems," in *PRDC*. IEEE, 2012, pp. 61–69.

[3] M. Machin, F. Dufossé, J.-P. Blanquart, J. Guiochet, D. Powell, and H. Waeselynck, "Specifying safety monitors for autonomous systems," in *SAFECOMP*. LNCS, 2014.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*. Springer, 2002, pp. 359–364.

[5] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-Tiga: Time for playing games!" in *Computer Aided Verification*. Springer, 2007, pp. 121–125.

[6] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, 2009.

[7] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[8] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL - a tool suite for automatic verification of real-time systems," in *Hybrid Systems III*. LNCS, 1996, pp. 232–243.

[9] A. Pnueli, E. Asarin, O. Maler, and J. Sifakis, "Controller synthesis for timed automata," in *Proc. SSC. Elsevier*, 1998.

[10] H. Kress-Gazit, G. Fainekos, and G. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Transactions on Robotics*, vol. 25, no. 6, pp. 1370–1381, Dec 2009.

[11] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM journal on control and optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[12] L. Fotoohi and A. Graser, "Building a safe care-providing robot," in *ICORR*. IEEE, June 2011, pp. 1–6.

[13] R. D. McKelvey, A. M. McLennan, and T. L. Turocy, "Gambit: Software tools for game theory," 2006.

[14] P. Roy, P. Tabuada, and R. Majumdar, "Pessoa 2.0: a controller synthesis tool for cyber-physical systems," in *HSCC*. ACM, 2011, pp. 315–316.

[15] R. Alur, T. A. Henzinger, F. Y. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran, "Mocha: Modularity in model checking," in *Computer Aided Verification*. Springer, 1998, pp. 521–525.

[16] K. Akesson, M. Fabian, H. Flordal, and R. Malik, "Supremica-an integrated environment for verification, synthesis and simulation of discrete event systems," in *WODES*. IEEE, 2006, pp. 384–385.