SkiNet, A Petri Net Generation Tool for the Verification of Skillset-based Autonomous Systems

Baptiste Pelletier, Charles Lesire, David Doose, Karen Godary-Dejean, Charles Dramé-Maigné

SkiNet
A Petri Net Generation Tool for the Verification of Skillset-based Autonomous Systems

Baptiste Pelletier
ONERA/DTIS
Université de Toulouse, France
LIRMM
Univ. Montpellier, CNRS France
baptiste.pelletier@onera.fr

Charles Lesire
ONERA, DTIS
Toulouse, France
firstname.lastname@onera.fr

David Doose
ONERA, DTIS
Toulouse, France
firstname.lastname@onera.fr

Karen Godary-Dejean
LIRMM, Univ. de Montpellier, CNRS
Montpellier, France
karen.godary-dejean@umontpellier.fr

Charles Dramé-Maigné
LIRMM, Univ. de Montpellier, CNRS
Montpellier, France
charles.drame-maigne@ens-paris-saclay.fr

The need for high-level autonomy and robustness of autonomous systems for missions in dynamic and remote environment has pushed developers to come up with new software architectures. A common architecture style is to summarize the capabilities of the robotic system into elementary actions, called skills, on top of which a skill management layer is implemented to structure, test and control the functional layer. However, current available verification tools only provide either mission-specific verification or verification on a model that does not replicate the actual execution of the system, which makes it difficult to ensure its robustness to unexpected events. To that end, a tool, SkiNet, has been developed to transform the skill-based architecture of a system into a Petri net modeling the state-machine behaviors of the skills and the resources they handle. The Petri net allows the use of model-checking, such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL), for the user to analyze and verify the model of the system.

1 Introduction

The use of autonomous systems has spread widely in the recent years, with applications in industrial automation, scientific exploration, rescue or environment monitoring missions. They are able of executing critical tasks and missions that need a high level of dependability.

For the sake of improving the trust in the autonomy of robotic systems, different validation and verification methods (V&V) can be used, such as formal methods, like model-checking, to verify that user-defined properties will be guaranteed once the system is deployed. The verification of the system behavior whatever happens requires to predict all possible system configurations, failure points and their appropriate responses, as well as the complexity of a dynamic, sometimes unknown environment. For a system to answer each of these challenges, hardware and software complexity must increase, which makes the modelling and analysis processes more difficult. Using formal methods and verification tools such as model-checking, it is possible to prove the robustness of the system throughout its design stages.

Various software control architectures have been developed with always the same objectives in mind: making the system easier to specify, and its robustness easier to verify. Looking at the system with a higher-level of abstraction can make the verification process easier for users with few experience in

M. Luckcuck and M. Farrell; M. Autili, L. Berardinelli,
A. Bucaioni and C. Pompilio (Eds.):
Formal Methods for Autonomous Systems and
Automated and verifiable Software sYstem DEvelopment
EPTCS 371, 2022, pp. 120–138, doi:10.4204/EPTCS.371.9

middleware and low-level architecture. But abstraction becomes quickly limited when facing dynamic environments and complex tasks. An ideal model would then be one that is abstract enough to allow an easy specification, while also providing great verification possibilities, which is the goal of the present study.

The software architectures tackled in this work are skill-based architectures, also called task-based [21, 1, 30], where the robotic system is decomposed into elementary actions. This approach usually comes with a high-level of abstraction of the software and hardware of the system, while offering a wide range of specification possibilities. A controller layer is then implemented, to test the architecture and control the system. The architecture used for this work will be the Skillset formalization of Albore et al. [1], which is composed of resources, skills and events and can model the robot as well as its environment and operator decisions. The strength of this formalization is that a code generating tool is provided, which will guarantee that the execution of the system will be as specified in the skillset.

The present work was conducted with the objective of improving the current verification capabilities of the skillset architecture by using a Petri net equivalent to the skillset model. The long-term objective is to offer an accessible model-checking tool for this architecture, to ensure that all people involved in the design of a robotic system can understand and contribute to the verification process with their own technical knowledge. Indeed, current autonomous systems are becoming more costly and multidisciplinary, while being used in remote and hazardous environments, where human assistance is nearly impossible. This requires scientific inputs during the design phase from researchers requesting valuable data, field experts that will put physical limitations to the system, and technical operators that will design the robotic hardware and software. In order for all these participants to contribute, the verification process must be simplified. For instance, formalisms such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) have important capabilities to verify the behavior of a system, but demand a solid knowledge in the functioning of model-checking in order to use them properly.

Various formal methods are used for the purpose of property verification, and the present tool we present in this paper, SkiNet, uses Petri nets to translate the skillset model and perform model-checking indirectly. The end user never manipulates the Petri net nor the model-checking tools, as SkiNet performs all these steps in the background and only provides to the user the valuable informations for the skillset design process. This is especially useful as Petri nets modelling complex systems can become difficult to read visually, and properties to check also become proportionally more complex.

Thus, this paper presents SkiNet and the methods it uses. Section 2 will begin by looking at the related work tackling the verification of high-level software architectures using formal methods. Then, section 3 will sum up the background, with the definitions of Petri nets and the skillset architecture. Section 4 will go over the content of the generated Petri net and the generation process. Section 5 will prove the correctness of the generated Petri net with regards to the state-machine behavior of skills and resources. Finally, section 6 will show a few examples of model-checking done on the generated Petri net to verify the behavior of the skillset model, before moving on to the conclusion in section 7. An illustrating example with the Boston Dynamics Spot® robot, shown in Fig. 1, with the `spot` skillset, in Fig. 2, used for generating the controller code. The SkiNet tool and instructions are available publicly for the readers to try it out themselves: `https://gitlab.com/onera-robot-skills/skinet-release`.

2 Related works

To facilitate the development of autonomous systems, skill-based architectures have been designed to decompose the system into elementary actions. An early take on the subject was done in [21], with many

new concepts emerging with an increasing demand for autonomous systems from industries [30]. The main advantage of such architectures are their modularity, easy reconfiguration and repurposability, as well as making robotic programming easier for end users. The skills are then composed to create more complex tasks to achieve specific goals or missions, while using feedback from sensors and actuators [30, 1, 36, 35, 18]. The mission design using skill-based architecture is often coupled with deliberative functions to create autonomous systems capable of adapting to their environment or faulty behaviors, often using model-based architectures [16, 29]. This is a growing need in domains such as underground, underwater [23, 31, 38] or space exploration [13, 14]. However, such functions imply a precise specification of all possible faults that can arise, for instance by designing fault-trees [12], and implementing a proper risk-management architecture [24, 2].

In order to prepare the autonomous system for high-risk missions and guarantee a-priori its robustness to the dynamic environment it will evolve in and to the faults that could arise, formal methods based on model-checking have been developed, sometimes at a very early development stage. Formal methods are used to tackle the verification of either the mission design or the entire robotic system. Ingrand [15] gathers the state of the art of V&V formal methods for autonomous systems. Albore [1] shows how missions can be designed through Behavior Trees, allowing for robust mission specification and fault management, however such method restricts the verification of the system to a specific mission. Evans [8] proposes a Model-Based Mission Assurance approach to improve the safety and robustness of a system in the early development stage, using Assurance models synthesis, implementing fault trees and Bayesian nets as inputs for SysML diagrams, opening interesting perspectives for the implementation of fault management into software architecture. Nardone [27] proposes a methodology for the V&V of satellite operational mode management specification, using mu-calculus logic. The method offers an interesting insight on the use of mu-calculus logic for V&V. Gross [11] also uses formal specification for the early stages of spacecraft design and attitude control system and shows the cost-reducing capabilities of using model-checking in a large-scale project. Finally, Louis [20] designed a mission controller based on a Fault Management System for an underwater automated vehicle. This approach, while being very safe with regards to system integrity, also demands the inputs from experts to specify the fault model and calibrate the controller correctly.

Petri nets [32, 26] are used in a wide range of applications such as robotics, industrial management or video-games [33], with a wide range of tools readily available to manipulate and verify Petri nets, such as the Tina toolbox (Time Petri net Analyzer) [4, 5], and thus indirectly verify the modeled system or mission [18, 7, 25]. Costelha [6, 7] uses Petri nets to model and verify robotic tasks. This early approach is very close to our goal here: a design-analysis-design approach that improves user experience when designing the robot model before using the real robot. However, the nets were manually built, while the present work is based on written specification synthesis. Reza [34] uses both Petri nets and converted Fault trees into nets for verification and safety analysis for systems based on Requirements State Machine Language (RSML). This approach is similar to the present work, with automatic synthesis of RSML into fault trees and Petri nets. A similar approach was proposed by Yan [37] for autonomous mission reliability modelling. Kwon [17] uses specialized colored Petri nets to model context-aware agent-based applications. The systems are decomposed into context-independent patterns, which is an approach also considered by Figat [9], with hierarchical Petri nets, where layered patterns are used to describe components of multi-agent robotic systems. These approach show how much complexity can be put into Petri nets to model a system, and how easily customizable they are to suit the modelling need of every user and/or system. Multiple agents missions verification is also tackled by [28], and such approaches open future perspectives on the present work for the use of skill-architecture based Petri nets with multiple coordinating agents. Finally, Mahulea [22] proposes an automatic generation of Petri nets

for boolean based robot planning, with generation and verification processes also similar to the present work, but without the skill-based architecture context.

Finally, controller synthesis using formal methods has been widely developed to close the gap between a safe model and a controller code that respects this model, while relieving the programmers from writing their own middleware. Foughali proposes the use of GenoM3 [10], a framework to specify robotic systems in the form of component-based timed transition systems, from which controller synthesis can be performed using templates. GenoM3 uses a lower-level approach than the skillset architecture presented in this work, but raises critical points on the usability of controller synthesis frameworks with other existing V&V tools. Figat [9] suggests hierarchical Petri nets to decompose multi-agents systems into functional layers. This hierarchical perspective allows for the use of ready-to-use templates to model low-level components, so that the user can focus on modelling and verifying the higher-level architecture. Controller synthesis is also performed using this framework, however the Petri nets are hand-written and not synthesized from user-written specification, which can make it difficult for a non-experienced user to manipulate. Lesire [18] creates a Skills Colored Petri net controller and synthesizes it into a ROS node to manage the execution of the system. Unfortunately, the resources system was not yet developed, which limited mission complexity. On top of this, Colored Petri nets are a special formalization of Petri nets, which limits the available tools for verification.

The present work uses the same framework as [1], where skillset specifications are written by the user, and a skill-manager layer is synthesized into a ROS node to complement with pre-built system control functions. Our work here is to provide a verification tool for this architecture, using Petri nets. The goal of the tool is to translate the user-written specification of the skillset into a Petri net, by using the definition of skill-nets presented in [18], with the addition of the resources system and events. We also removed the colored net formalism to be able to use more verification tools. Controller code can be generated from the skillset specification, and our tool can be used for V&V purpose, during both design and deployment phase, thus fully covering the aforementioned problematics of system development.

3 Background

3.1 Petri net

This section will sum up the definitions and notions of the generated Petri net, as defined by [32, 26], with the addition of transition priority as presented by Balbo [3].

Definition 3.1 (Petri net)**.** A Petri net $\langle N, m_0 \rangle$ is a tuple $N = (P, T, F)$ and an initial marking m_0, where:

- P and T are two non-empty, disjoint and finite sets of places and transitions, respectively.
- $F \subseteq (P \times T) \cup (T \times P)$ a set of directed arcs.
- $m_0 \in M$ the initial distribution of tokens, called the initial marking of the net, and $M = \{m_0, m_1, ..., m_n\}$ the set of all possible markings of N.

This definition is extended with $N = (P, T, F, \succ)$, where $\succ$ is the priority relation, represented by directed arcs between transitions, with the source transition having a higher priority. This means that if two transitions $t_1, t_2 \in T$ are enabled, i.e. their input places have at least one token, and $t_1 \succ t_2$, then only t_1 is firable.

For any transition $t \in T$, the sets of its input and output nodes are ${}^\bullet t$ and $t^\bullet$ respectively. While arcs are usually weighted in conventional Petri nets, only unitary arcs will appear in this paper, with an explanation in section 4.

Let $p \in P$ and $t \in T$ be a place and a transition. The marking of a place p is noted $m[p]$. The firing of an enabled transition $t \in T$, with $\forall p \in^\bullet t,\ m[p] \geq 1$, leads to a new marking, or reachable state, m'. All the input places of t loose a token, i.e. $\forall p \in^\bullet t,\ m'[p] = m[p] - 1$, and all the output places gain one token, i.e. $\forall p \in t^\bullet,\ m'[p] = m[p] + 1$.

3.2 Skillset architecture

This section summarizes the elements of a Skillset used for the modelling and programming of autonomous systems, as defined by Albore [1], with some elements omitted as they are not used in this paper. The Skillset can represent both hardware and software elements of the system, and their interaction/execution. A tool called "robot language" generates C++ code based on the skillset specifications that follows this execution, with part of the execution code to be filled by the user, such as skills functions, exit conditions, events triggering, etc. More information on the skillset execution semantic can be found in [1]. A skillset contains resources, resource guards and resource effects. These elements are assembled into skillset transitions, to create events and skills. We define all the elements of a skillset in the following section. An example skillset is given in 2, which was used for generating controller code for the Boston Dynamics Spot® robot, a quadruped robot capable of carrying heavy payload and perform observation tasks, shown in Fig. 1. This example showcases the syntax of the skillset specification language as it would be written by the user, which is significantly simpler than the underlying formal definition, given in the following section.

Figure 1: The Spot® quadruped robot, mounted with extra payload. The controller code was generated from a skillset which was verified a-priori with SkiNet. The skillset in Fig 2 is an extract of the actual running skillset, available at `https://gitlab.com/onera-robot-skills/skinet-release`.

3.2.1 Resources

Definition 3.2 (Resources)**.** Resources are state-machines that represent the status of an element of the system (sensor, motorization status, mutex...). A resource r is a tuple (S^r, T^r), with a set of states $S^r =$

```
skillset spot{

    resource {
        power_status {
            initial PowerOff
            PowerOff -> PowerOn
            PowerOn -> PowerOff
        }
        lease_status {
            initial AutoMode
            AutoMode -> ManualMode
            ManualMode -> AutoMode
        }
        control_mode {
            initial Idle
            Idle -> Busy
            Busy -> Idle
        }
    }

    event {
        toauto_frommanual {
            guard lease_status == ManualMode
            lease_status -> AutoMode
        }
        tomanual_fromauto {
            guard lease_status == AutoMode
            lease_status -> ManualMode
        }
        power_switchoff {
            guard power_status == PowerOn
            power_status -> PowerOff
        }
        power_switchon {
            guard power_status == PowerOff
            power_status -> PowerOn
        }
    }

    skill init_power {
        precondition {
            canmove {
                guard lease_status == AutoMode
                and control_mode == Idle
            }
            ispowered {
                guard power_status == PowerOff
            }
        }
        start control_mode -> Busy
        invariant {
            is_busy {
                guard control_mode == Busy
            }
        }
        success is_poweredon {
            control_mode -> Idle
            power_status -> PowerOn
        }
        failure couldnot_power control_mode -> Idle
    }

    skill safe_poweroff {
        precondition {
            canmove {
                guard lease_status == AutoMode
                and control_mode == Idle
            }
            ispowered {
                guard power_status == PowerOn
            }
        }
        start control_mode -> Busy
        invariant {
            is_busy {
                guard control_mode == Busy
            }
        }
        success is_poweredoff {
            control_mode -> Idle
            power_status -> PowerOff
        }
        failure couldnot_poweroff control_mode -> Idle
    }

    skill go_to{
        precondition {
            canmove {
                guard lease_status == AutoMode
                and control_mode == Idle
            }
            ispowered {
                guard power_status == PowerOn
            }
        }
        start control_mode -> Busy
        invariant {
            is_auto {
                guard lease_status == AutoMode
            }
            is_powered {
                guard power_status == PowerOn
            }
        }
        interrupt control_mode -> Idle
        success is_arrived control_mode -> Idle
        failure not_arrived control_mode -> Idle
    }

}
```

Figure 2: Extract of the `spot` skillset model for the Boston Dynamics Spot® robot. Resources model motor power state, lease mode and control mode mutex. Events are used to represent the actions of the operator on the robot. Skills the robot can perform are `init_power` and `safe_poweroff` to start and cut motor power respectively, and a movement skill `go_to`.

$\{S_0^r, ..., S_{n-1}^r\}$, and a set of transitions between each state $T^r \subset (S^r \times S^r)$. A resource can only be in one of its states at any given time during the execution of a skillset, noted $state(r) \in S^r$, with S_0^r the initial state.

The set of states that can lead to a state S_i^r and the set of states that can originate from S_i^r are noted ${}^\bullet S_i^r$ and $S_i^{r\bullet}$ respectively. The transition from S_i^r to itself is a valid transition.

Definition 3.3 (Resources effects and guards)**.** Resource effects aim to change the state of a specific resource. An effect ε is a tuple (r, S_i^r) of a resource and its next state. The origin state of the resource is not specified, as effects only mention the destination state. Effects can be empty and are only valid if

they contain at most one effect for each resource.

Resource guards are used to put conditions on the triggering of effects. A resource guard, or simply guard, is a logical formula $\phi : \{S^r, r \in \mathscr{R}\} \rightarrow \{True, False\}$ on the states of the resources of $\mathscr{R}$.

3.2.2 Skillset transitions

The skillset is then assembled by coupling guards and effects to create skillset transitions. Skillset transitions are used in events and skills, and represent the basis of the execution of the skillset. We also define solutions, the set of resource states that verify a resource guard.

Definition 3.4 (Skillset transitions)**.** A skillset transition is a tuple $\tau = (\phi, \mathscr{E}, \sigma)$, formed with a guard ϕ, a set of effect $\mathscr{E}$, and a state change σ. A state change is a tuple $\sigma = (state1, state2)$ which changes the state of a skill, later defined in Def 3.6. We note $\mathscr{R}(\tau)$ the resources involved in either ϕ or $\mathscr{E}$. A skillset transition can be triggered if its guard is true. If triggered, its effects and state change are applied.

Definition 3.5 (Solutions)**.** X_τ is the set of all solutions of a skillset transition τ. A solution is a set of resource states $x = \{x_r \in S^r, r \in \mathscr{R}(\tau)\}$ associated to a guard ϕ, such that $\phi(x) = True$. x only contains up to one state per resource. The solutions X_τ can be obtained with a boolean satisfaction problem solver (SAT solver).

3.2.3 Skillset

Definition 3.6 (Skillset)**.** A skillset is a tuple $\Sigma = (\mathscr{R}, \mathscr{V}, \mathscr{S})$ with:

- $\mathscr{R}$ is a set of resources $\{r_1, r_2, ...\}$.
- $\mathscr{V}$ is a set of events. An event is a skillset transition τ_v with state change $\sigma = \varnothing$. Events represent actions from the exterior of the system, such as from the operator or the environment, on the resources of the skillset. They can have a guard or none.
- $\mathscr{S}$ is a set of skills, which are elementary actions that can be executed by the system. A skill $s \in \mathscr{S}$ is a tuple $s = (name, pre, inv, state, \mathscr{T}_s)$, with:
  - An identifier *name*.
  - Preconditions $pre = \{pre_1, pre_2, ...\}$ are a set of guards that must be verified to start the skill. If they are not satisfied while attempting to start the skill, we may have failure effects $\bar{pre}_i$ triggered on each unsatisfied guard.
  - $inv = \{inv_1, inv_2, ...\}$ is a set of guards, called invariants, that must remain true during the skill execution. An invariant failure results in an immediate termination of the skill and in the application of the failure effects $\bar{inv}_i$.
  - A set of states *state*: idle e_s, running i_s or terminated $x_{s,k}$, with the termination mode $k \in \{\bar{pre}, \bar{inv}, succ, fail, int\}$. This notation is adapted from the colored Petri net formalism used by Lesire et al. [18]. At the initial state of the skillset, all skills are idle.
  - A set of skillset transitions $\mathscr{T}_s = \{\tau_{s,start}, \tau_{s,\bar{pre},i}, \tau_{s,\bar{inv},i}, \tau_{s,succ}, \tau_{s,fail}, \tau_{s,int}\}$, with:
    - Start $\tau_{s,start} = \{pre, start, e_s \rightarrow i_s\}$, with guards $pre = pre_1 \wedge pre_2 \wedge ...$, start effects *start* and state change $e_s \rightarrow i_s$. $\tau_{s,start}$ is triggered when a skill is started and all preconditions *pre* are verified. The *start* effects are then applied and the skill goes from idle e_s to running i_s.

* Precondition failures $\tau_{s,\bar{pre},i} = \{\neg pre_i, \bar{pre}_i, e_s \rightarrow x_{s,\bar{pre},i}\}$ are triggered if the start of a skill is attempted but the precondition pre_i is not verified, i.e. resources states do not verify the resource guard.
* Invariant failures $\tau_{s,\bar{inv},i} = \{\neg inv_i, \bar{inv}_i, i_s \rightarrow x_{s,\bar{inv},i}\}$ are triggered if the skill is running and the invariant inv_i is not verified.
* Success $\tau_{s,succ} = \{inv, succ, i_s \rightarrow x_{s,succ}\}$, failure $\tau_{s,fail} = \{inv, fail, i_s \rightarrow x_{s,fail}\}$ and interrupt $\tau_{s,int} = \{inv, int, i_s \rightarrow x_{s,int}\}$, are triggered when the invariants *inv* are still verified and the skill execution ends with a success, failure or interrupt respectively.

More detail on the skill execution state-machine can be found in [1]. Note that multiple successes and failures can be added, but only one interrupt is possible.

Using the `spot` skillset example from Fig. 2, the execution of a skill is as follows: in order to perform the `go_to` skill, the preconditions `canmove` and `ispowered` must be satisfied. In that case, starting the skill will trigger the *start* effects. Upon starting the skill, invariants satisfaction is checked. While the guards `is_auto` and `is_powered` are respected, the skill is running, until the skill ends, triggering the associated effects *int*, *succ* or $fail$.

4 SkiNet Architecture

SkiNet was designed with the aim to translate the skillset of a robotic system into an equivalent Petri net that would reproduce its behavior, in order to perform model-checking using Petri net verification tools, and verify the behavior of the skillset indirectly. A graphical representation of SkiNet in its context is available in Fig. 3. In this section, the translation process of a skillset $\Sigma = (\mathscr{R}, \mathscr{V}, \mathscr{S})$ into a Petri net $N = (P, T, F, \succ)$ will be explained, while section 5 will prove the correctness of the translation, and section 6 will show the verification capabilities of the tool using the generated Petri net.

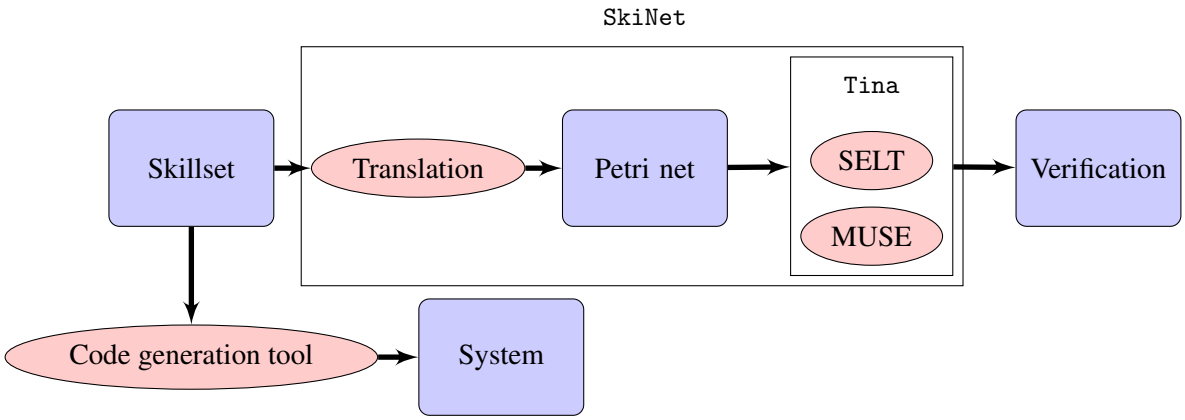

Figure 3: SkiNet is a tool used to generate a Petri net from a skillset, on which model-checking can be performed, using tools taken from Tina [4, 5]. SkiNet acts as a layer to avoid manipulating the tools directly. The verified skillset can then be used for controller code generation.

In this work, we use the foundations of skill Petri nets presented by Lesire et al. [18]. However, the colored Petri net formalism used has been removed. While it allowed the net to be more expressive, colored Petri nets don't have the same range of verification tools as classical Petri nets, such as the Tina

toolbox [5]. The expressivity lost doing so is not relevant here, thanks to SkiNet which extracts the useful informations from the generated Petri net without the need for the user to look at it. Moreover, colored Petri nets only make the nets more compact, but do not reduce the size of the marking space M, so the complexity is identical.

4.1 Places of the generated Petri net

The places of the generated Petri net are $P = P_r \cup P_s$, with:

- $P_r = \{p_i^r, 0 \leq i \leq |S^r|, r \in \mathscr{R}\}$ the resource state places, with a place p_i^r for each state $S_i^r \in S^r$ of each resource r of the skillset Σ.
- $P_s = \{p_e^s, p_i^s, p_{x,k}^s, k \in \{\bar{pre}, \bar{inv}, succ, fail, int\}, s \in \mathscr{S}\}$ for $\mathscr{S} \in \Sigma$, the places representing the skills execution state. This representation is inspired from the work of Lesire et al. [18] and defined as follows:
  - p_e^s is the entry place. This place must be marked, i.e. $m[p_e^s] \geq 1$ for the skill s to start.
  - p_i^s is the intermediate place. The start of the execution of the skill s is represented by the start transition which will move the token from p_e^s to p_i^s.
  - $p_{x,k}^s$ are the exit places. Upon skill termination, the token is moved from p_i^s to the place $p_{x,k}^s$ of the corresponding termination mode k.

4.2 Transitions of the generated Petri net

For each skillset transition $\tau = (\phi, \mathscr{E}, \sigma)$, a set of transitions T_τ is generated in the Petri net N. For each solution $x \in X_\tau$ of ϕ, a transition $t_x \in T_\tau$ is generated, if and only if:

- The input places of t_x are the resource state places in x: for each $x_r \in x$, we have p_i^r the Petri net place of the resource state S_i^r, and $p_i^r \in {}^\bullet t_x$. Up to one resource state place p_i^r exists in ${}^\bullet t_x$ for each resource, as only one state per resource exists in each solution x, cf Def. 3.5.
- The output places of t_x are the destination states S_i^r of the effects $\varepsilon \in \mathscr{E}$: for $S_i^r \in \varepsilon, \varepsilon \in \mathscr{E}$, we note p_i^r the Petri net place of S_i^r, and $p_i^r \in t_x^\bullet$. Because there is at most one effect ε for each resource,cf Def. 3.3, then there is only at most one resource state place p_i^r for each resource.
- The input and outputs are extended with the states in $\sigma = (state1, state2)$. We note p_1^s and p_2^s the skill state places of each state in σ, and we have: $\forall t_x \in T_\tau, p_1^s \in {}^\bullet t_x$ and $p_2^s \in t_x^\bullet$. If $\sigma = \varnothing$, then no place is added to the inputs and outputs of t_x.

The generation of the transitions needs to be further complexified in order to overcome the limits of Petri nets. Indeed, there are cases of skillset transitions τ where a resource is evaluated by the guard ϕ but not present in the effects $\mathscr{E}$, and vice-versa. Therefore, we need to generate the transitions according to these issues:

- If a resource is *nominal*, i.e. both guarded and affected in τ, then nothing needs to be done on T_τ. The generated transitions simply move the tokens between the states of the resource as already specified in τ.
- If a resource is *unaffected*, i.e. guarded but with no effects in τ, then the corresponding state place $p_x^r \in {}^\bullet t_x$ would lose a token upon firing t_x, and this token would not be restituted. This would mean that the resource could be in an empty state, which is impossible. Therefore the token must be returned to the same state after firing t_x, i.e. $t_x^\bullet \leftarrow t_x^\bullet + p_x^r$.

- If a resource is *unguarded*, i.e. affected but not guarded in τ, then the corresponding destination state place $p_y^r \in t_x$ would receive a token, without first taking one from the state places of the resource. This would mean that the resource could be in two states at the same time, which is also impossible. Therefore, the token must be taken somewhere among the state places p_i^r of the resource. However, we cannot anticipate where the token is for a given marking, therefore all possible resource state changes must be computed, given the transition exists among the valid transitions of T^r that lead to S_y^r, i.e. ${}^\bullet S_y^r$, cf Def. 3.2. We note X_r the set of the state places p_i^r corresponding to the resource states in ${}^\bullet S_y^r$, and we extend X_τ as: $X_\tau \leftarrow X_\tau \cup X_r$.

After building all the transitions, we have $\bigcup_\tau T_\tau$ the set of all transitions as defined previously. To this set, we add a reset transitions set T_{reset}, to allow for the repeatability of skills, which moves the token from the exit places $p_{x,k}^s$ to the entry place p_e^s, with one reset transition per exit place, i.e.:

$$\forall s \in \mathscr{S}, \exists t_{s,reset,k} \in T_{reset}, {}^\bullet t_{s,reset,k} = p_{x,k}^s \tag{1}$$

$$t_{s,reset,k}^\bullet = p_e^s, k \in \{p\bar{re}, i\bar{nv}, succ, fail, int\} \tag{2}$$

The final set of transitions in the generated Petri net is then $T = (\bigcup_\tau T_\tau) \cup T_{reset}$.

Finally, transitions of invariant failures $T_{i\bar{nv}} = \bigcup T_{\tau,i\bar{nv}}$ have a higher priority than all other transitions, i.e.:

$$\forall t_{i\bar{nv}} \in T_{i\bar{nv}}, \forall t \in T - T_{i\bar{nv}} : t_{i\bar{nv}} \succ t \tag{3}$$

This allows to respect the execution semantic of the skills, and forces a skill to terminate first if its invariant is broken before firing another transition, which could potentially "repair" the invariant.

4.3 Examples with the `spot` skillset model

Using the skillset model of Spot® shown in Fig 2, the generation process of two transitions, $\tau_{go\_to,start}$ and $\tau_{go\_to,sucess,is\_arrived}$ of the skill `go_to`, will be presented, and the results shown in Fig. 4.

- Start transition :

$$\begin{aligned} \tau_{go\_to,start} = \{ & \phi = \{(\texttt{lease\_status} == AutoMode \wedge \texttt{control\_mode} == Idle) \\ & \wedge \texttt{power\_status} == PowerOn\}, \\ & \mathscr{E} = \{\texttt{control\_mode} -> Busy\}, \\ & \sigma = (e_{go\_to}, i_{go\_to})\} \end{aligned} \tag{4}$$

The guard ϕ of this transition is the product of the preconditions `canmove` and `ispowered`, while the effects set $\mathscr{E}$ contains the start effect. The set of solutions that satisfies ϕ contains only one element:

$$X_\tau = \{(AutoMode, Idle, PowerOn)\} \tag{5}$$

The resource states in x will be used to build the input places of the start transition. Because `power_status` and `lease_status` are unaffected, the places *PowerOn* and *AutoMode* are both inputs and outputs, in order to restitute the tokens. The resource `control_mode` is nominal as it is both guarded and affected. The transition also moves the token of the skill state places from the idle place $p_e^{go\_to}$ to the running place $p_i^{go\_to}$. In conclusion, only one transition is generated, shown in Fig. 4:

$$T_{\tau_{go\_to,start}} = \{\texttt{t\_start\_go\_to}\} \tag{6}$$

- Success transition :

$$\begin{aligned}\tau_{go\_to,success\_is\_arrived} = \{&\phi = \{\texttt{lease\_status} == AutoMode \wedge \texttt{power\_status} == PowerOn\},\\ &\mathscr{E} = \{\texttt{control\_mode} -> Idle\},\\ &\sigma = (i_{go\_to}, x_{go\_to,success,is\_arrived})\}\end{aligned} \tag{7}$$

The guard ϕ is the product of the invariants, here `is_auto` and `is_powered`, and the effects set $\mathscr{E}$ contains the success effect. The set of solutions that satisfies ϕ again contains only one element:

$$X_\tau = \{(AutoMode, PowerOn)\} \tag{8}$$

The resource `lease_mode` is again unaffected, so the state that solves ϕ, *AutoMode*, is both input and output of the transitions. `control_mode` is unguarded this time, so two transitions are needed in order to account for all possible states of the resource that can lead to *Idle*, which are *Idle* and *Busy*. So one transition moves will move the token from *Idle* to *Idle*, and the other from *Idle* to *Busy*. Finally, the token of the skill state places is moved from $p_i^{go\_to}$ to the exit place $p^{go\_to}_{x,success,is\_arrived}$. The generated transition are then, as shown in Fig. 4:

$$\begin{aligned}T_{\tau_{go\_to,success\_is\_arrived}} = \{&\texttt{t\_go\_to\_success\_is\_arrived\_0},\\ &\texttt{t\_go\_to\_success\_is\_arrived\_1}\}\end{aligned} \tag{9}$$

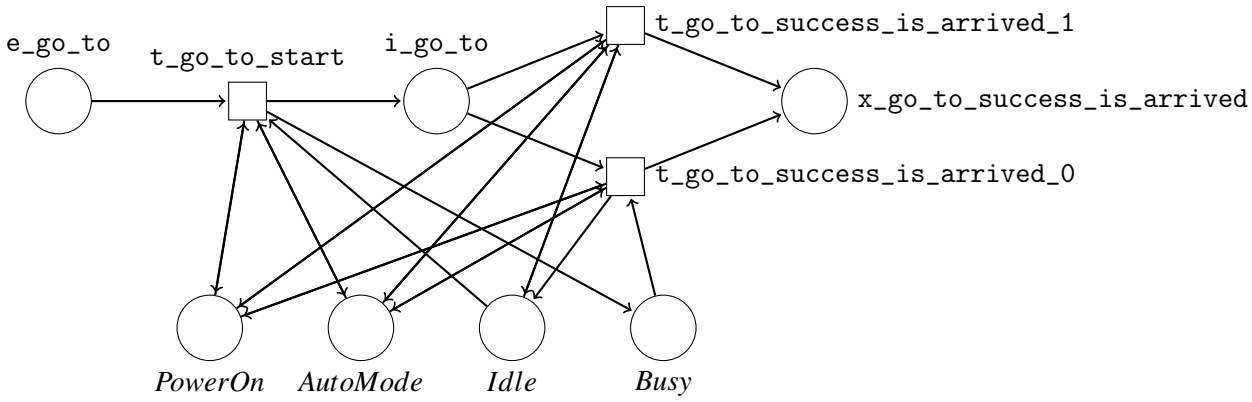

Figure 4: Start and success transitions of the skill `go_to` as generated by SkiNet. `e_go_to`, `i_go_to` and `x_go_to_success_is_arrived` are the internal places of the skill, representing the Idle, Running and Ended states respectively. *PowerOn*, *AutoMode*, *Idle* and *Busy* are the places representing the states of the resources used with the skill.

4.4 Initial Marking

The initial marking of the net follows the initial state of the skillset. Resources are in their initial state, so only this place has a token, i.e:

$$r \in \mathscr{R}, p_0^r = 1, p_{i\neq 0}^r = 0 \tag{10}$$

Skills are all idle at the initial state, so we have:

$$s \in \mathscr{S}, p_e^s = 1, p_i^s = 0, \forall k, p_{x,k}^s = 0 \tag{11}$$

Because resources and skills states do not stack, i.e. a resource/skill cannot be twice in the same state at the same time, only one token must be present in a place for any marking. This is called 1-safeness:

$$\forall m \in M, \forall p \in P : m[p] \in \{0,1\} \tag{12}$$

5 Approach Validation

In this section, the validation of the translation process presented in section 4 will be conducted, using mathematical induction, to prove that the generated Petri net satisfies the state-machine properties of resources and skills. First, the two properties to verify will be presented, and their validity at the initial marking will be proven. Then, the induction for any marking will be developed, before concluding on the validity of the properties.

5.1 Place invariants of state-machines

The resources places and skills state places must share only one token at all time. It is critical that the generated transitions will satisfy the two following properties:

Proposition 1 (Invariants)**.** For any marking of the net, i.e. $\forall m_n \in M$ the following equations are satisfied:

$$\forall r \in \mathscr{R}, \sum_{S_i^r \in S^r} m_n[p_i^r] = 1 \tag{13}$$

$$\forall s \in \mathscr{S}, \; m_n[p_e^s] + m_n[p_i^s] + \sum_k m_n[p_{x,k}^s] = 1 \tag{14}$$

By extension of this definition, because the set of places P of the generated Petri net is only composed of the resources and skills places, the safeness of the net, Eq. (12), is proven if Eq. (13) and (14) are satisfied:

Lemma 1 (Safeness)**.** If Eq. (13) and(14) are satisfied, then the Petri net is 1-safe, i.e.:

$$\forall m \in M, \forall p \in P, m[p] \leq 1 \tag{15}$$

Let us begin the proof with the initial state.

Proof by induction - Initial state. At $n = 0$, $\forall r \in \mathscr{R}$, the state-machine of r is in the initial state $state(r) = S_0^r$, the initial marking of the net is $m_0[p_0^r] = 1$, and $m_0[p_{i\neq 0}^r] = 0$. This gives $\sum_{S_i^r \in S^r} m_0[p_i^r] = 1$ and satisfies equation (13) for any $r \in \mathscr{R}$.

Additionally, $\forall s \in \mathscr{S}$, all skills are in the idle state e_s, therefore $m_0[p_e^s] = 1$, $m_0[p_i^s] = 0$ and $\sum_k m_0[p_{x,k}^s] = 0$. This gives $m_0[p_e^s] + m_0[p_i^s] + \sum_k m_0[p_{x,k}^s] = 1$ and satisfies equation (14) for any $s \in \mathscr{S}$.

Therefore, the proposition is true for the initial marking m_0. □

Now, let us assume that, for a marking m_n, properties 1 and 2 are true. m_{n+1} is the state that follows the firing of any enabled transition $t \in T = (\bigcup_\tau T_\tau) \cup T_{reset}$. First, the reset transitions will be studied, then the event and skill transitions.

5.2 Reset transitions

Let us prove that for any firing of a reset transition, the properties (13) and (14) are still valid in the resulting marking.

Proof by induction - Reset transitions. If $t_{s,reset,k} \in T_{reset}$, a reset transition, is fired, then it will simply transfer the token from one of the termination state place $p^s_{x,k}$ to p^s_e, i.e. ${}^\bullet t_{s,reset,k} = \{p^s_{x,k}\}$ and $t^\bullet_{s,reset,k} = \{p^s_e\}$. Because equation (14) is true for m_n, we have:

$$m_n[p^s_{x,k}] = 1,\ m_n[p^s_{x,k'\neq k}] = 0,\ m_n[p^s_e] = 0, m_n[p^s_i] = 0 \tag{16}$$

Therefore, firing t will yield:

$$m_{n+1}[p^s_e] = 1,\ m_{n+1}[p^s_{x,k}] = 0,\ m_{n+1}[p^s_{x,k'\neq k}] = 0,\ m_{n+1}[p^s_i] = 0 \tag{17}$$

This holds Eq. (14), and because no resource is involved in the firing of $t_{s,reset,k}$, then we are sure that Eq. (13), which is true at m_n, will be true at m_{n+1}. We conclude that reset transitions hold both propositions as true when fired. □

5.3 Event and Skill transitions

Now, let us prove that for any firing of an event or skill transition, the properties (13) and (14) are valid in the resulting marking.

Proof by induction - Event and Skill transitions. If $t_x \in T_\tau$ is a transition generated from a skillset transition $\tau = (\phi, \mathscr{E}, \sigma)$, then we have:

- ${}^\bullet t_x = \{p^s_1, p^r_n, ...\}$ the input places composed of the state place p^s_1 and a unique resource state place p^r_n, corresponding to $state(r)$ at the marking m_n, for each resource $r \in \mathscr{R}(\tau)$.
- $t^\bullet_x = \{p^s_2, p^r_{n+1}, ...\}$ the output places composed of the state place p^s_2 and a unique resource state place p^r_{n+1} at the marking m_{n+1} for each $r \in \mathscr{R}(\tau)$.

Because Eq. (13) is true, for any $r \in \mathscr{R}(\tau)$, a single token is present in r at the state place p^r_n at the marking m_n, and the firing of t_x transfers the token to the state place p^r_{n+1} at the following marking m_{n+1}, i.e.:

$$\forall r \in \mathscr{R}(\tau),\ m_n[p^r_n] = 1,\ m_n[p^r_{i\neq n}] = 0 \tag{18}$$

And firing t leads to:

$$\forall r \in \mathscr{R}(\tau),\ m_{n+1}[p^r_{n+1}] = 1,\ m_{n+1}[p^r_n] = 0,\ m_{n+1}[p^r_j] = 0,\ j \notin \{n, n+1\} \tag{19}$$

Giving: $\forall r \in \mathscr{R},\ \sum_{S^r_i \in S^r} m_{n+1}[p^r_i] = 1$, so equation (13) is satisfied. □

N.B.: it is possible that the input state and destination places p^r_n and p^r_{n+1} are the same, since $S^r_i \in S^{r\bullet}_i, \forall S^r_i \in S^r$, as defined in Def. 3.2. In that case, the firing of t will take the token from p^r_n and restitute it, so if Eq. (13) was true at m_n, then it will be true at m_{n+1}.

For Eq. (14) , because t transfers the token from p^s_1 to p^s_2, for any skill $s \in \mathscr{S}$, and because Eq. (14) is true at m_n, then we can directly conclude that Eq. (14) will hold upon the firing of t as both x_s and y_s only contain one place among the internal places of s. Additionally, since $\sigma = \varnothing$ for events, then no skill state place is involved in the firing of t_x, therefore Eq. (14) is automatically verified at m_{n+1}.

Therefore, both properties (13) and (14) hold upon the firing of any $t \in T_\tau$. Finally, because the set of all transitions of the generated Petri net T is the union of the set of reset transitions T_{reset} and the events and skills transitions $\bigcup_\tau T_\tau$, then all transitions of T will hold the proposition.

5.4 Conclusion

Having proven that properties 1 and 2 are true at the initial marking m_0 of the generated Petri net, and that for any $m_n \in M$, the firing of any transition $t \in T$ holds the properties from m_n to m_{n+1} the marking reached by firing t, we can conclude that, for any marking m, the proposition is true.

This means that our generated Petri net respects the state-machine properties of resources and skills internal places, and that its behavior reflects correctly the behavior of the skillset as it would run on the autonomous system it models. We can now use the generated Petri net to do *a priori* verification by using Petri net analysis tools available in the litterature.

6 Model Verification

Now that the net generation aspect of SkiNet has been presented and validated, the possible model-checking thanks to the net will be presented in this section. SkiNet offers a simple interface to execute all or part of the following verification steps, without the need for the users to use the model-checking tools directly. However, it will still display the formulas used, for transparency, but also to guide the users if they wish to check their own properties with the same model-checking tools. The corresponding SkiNet verification name for each step is indicated in the subsection name, between quotation marks, and is tested on the `spot` skillset in Fig. 2. For reference, the computation time of the Petri net generation and its verification take less than a minute. Adding more resources and skills increases the calculation time, but so far no tested skillset was found to exceed a few minutes for the whole process. For flexibility purpose, options were added to remove events or exit places (tokens are directly moved to the entry place of skills upon firing an exit transition, and reset transitions are removed), to increase model-checking capabilities, computational time and overall scalability.

6.1 Kripke Structure of the skillset net

In order to perform model-checking using the tools Selt (LTL) and Muse (mu-calculus, CTL) of the Tina toolbox, we first need to convert the generated Petri net into a Kripke structure, on which temporal logic operations can be performed. Liu [19] summarizes the Kripke structure definition, generated, and later used as input, by the Tina toolbox. More informations and instructions can be found on the Tina toolbox website [5].

6.2 Petri net Deadlocks - "dead"

Deadlocks are markings where no transition is fireable. In this context, it means that no actions from skills or events can be done, and no reset transition is available. This can happen purposefully, for instance if a resource represents the battery state of the robot, and upon entering a critical state, all skills and events are blocked and nothing can happen anymore. However, it can also be the result of an error in the specifications of the skillset, so it's important to know what sequence of transitions led to the dead marking. In the tool Selt, the predicate *dead* has already been implemented to check if a state is a deadlock, therefore the LTL formula is to check for deadlocks in the net is "we never have a deadlock". Using the temporal operator "A" (Always):

$$A\neg dead \tag{20}$$

The program will either return *True* if no path leads to a deadlock, *False* if there is one, with a counter-example showing a sequence of transitions leading to the dead state. In the example skillset `spot` from Fig. 2, no deadlock was found, as the events can be fired infinitely, so there is always an event transition enabled.

6.3 Petri net Liveness - "live"

The non-dead transitions are searched. If a transition is never fireable, it is considered dead. Dead transitions appear almost in every case of net generation, and are due to the transition generation algorithm presented in section 4.2, which can find solutions that will never happen.

For the generated Petri net, a transition is not dead if it can be fired at least once, which can be checked with the formula:

$$\forall t \in T : A\neg t \tag{21}$$

If the formula is true, then the transition is dead. SkiNet will return the list of all dead transitions when running the "live" option. Looking at the `spot` skillset and the example success transition in Fig. 4, `t_go_to_success_is_arrived_1` was found to be a dead transition. This is because the resource `control_mode` is never in the state *Idle* during the execution of the skill `go_to_body`. This verification step can become quite time consuming for Petri nets with a large amount of markings, but the `spot` example is ran in only a few seconds as it is very simple.

6.4 Skillset Invariants verification - "safe"

In section 5, it was verified that the transitions generated respect the resources and skills places invariants, i.e. only one token exist at any marking in the same resource/skill places. In order to verify Eq. (13) and (14) using LTL, the following formula can be used for resources, which verifies that the sum of tokens among the places of the resource is always one:

$$\forall r \in \mathscr{R} : A(\sum_{S_i^r \in S^r} p_i^r = 1) \tag{22}$$

And for skills:

$$\forall s \in \mathscr{S} : A(p_e^s + p_i^s + \sum_k p_{x,k}^s) = 1) \tag{23}$$

Finally, the safeness of the net presented in Eq. (12) can also be checked using LTL, i.e. no place has more than one token for any marking:

$$\forall p \in P : A\neg(p >= 2) \tag{24}$$

These three LTL formulas were mostly used during the development of SkiNet to verify that the code would generate a proper net as defined in section 4 and 5.

6.5 Skills and Skillset deadlocks - "deadskill" and "deadset"

A property specific to the skillset architecture that can be verified is whether a skill, two concurring skills, or all the skills, are always alive or not. In other terms: is there a configuration in the skillset that can render a skill forever unusable. Here, LTL cannot be used, as these properties need to be checked for all paths, from any state. Therefore, mu-calculus and CTL are used with the Kripke structure, through the Muse tool of Tina.

The CTL property to check whether a skill is always eventually activatable is "For all paths globally, there is some path where finally p_i^s".

$$AGEF\, p_i^s, s \in S \tag{25}$$

This property either returns all the states if it is true, and none if there exist a single state where it is false. But because it returns no state, it does not provide sufficient feedback for the user. Therefore, in order to know the specific transition sequences leading to a state that would make the skill forever unactivatable, the property "For all paths finally, there are no paths where finally p_i^s" is used, so that the Muse tool returns exploitable counter-examples:

$$\neg AFEF\, p_i^s, s \in S \tag{26}$$

The states returned are then stored and the transition paths leading to these states can be obtained using the Pathto tool of Tina and the Kripke structure. For now, only the first of these states is returned by SkiNet. If the list of states is empty, then the skill can always be activated. It is possible to check for all skills at once with the property:

$$\neg AFEF(\sum_{s \in S} p_i^s) \tag{27}$$

In our example `spot`, an error was found in the skillset using these properties. SkiNet pointed out that, when executing the `go_to` skill, the failure of invariants `is_auto` and `is_powered` would block all the other skills from executing as the resource `control_mode` would not go back to the *Idle* state upon exiting `go_to`, thus blocking the precondition `canmove` of the skills. This was fixed by adding a failure effect `control_mode -> Idle` to the invariants (not shown in the Fig. 2).

7 Conclusion

The work presented in this paper aim to propose a method to verify the behavior of a software control architecture for critical autonomous systems. A tool was developed, called SkiNet, which generates a Petri net that models the behavior of the system as specified in the skillset model. Using the generated Petri net, SkiNet allows the users to check their skillset model by using model-checking. Basic properties such as Petri net deadlocks or transition liveness can be checked, as well as more skillset-specific properties such as skill liveness/deadlocks. Because the skillset-architecture upon which this work is based on is also a work-in-progress, future works will focus on proving the equivalence between the skillset semantic during runtime and the generated Petri net semantic, once the former is well established. Additionally, with always the same focus of making formal verification accessible to all actors of a robotic project, regardless of their background, two extensions of SkiNet are under progres. SkiNet Mission will allow users to create and verify the feasability of their missions, using Petri nets and model-checking, and SkiNet Live will offer users a view of the transition firing in the Petri net, synchronized with the skillset as it runs on the robotic system, for runtime analysis. The checking of properties during runtime is also being considered for the latter. Among the difficulties encountered, the complexity of the state space analysis of the generated Petri net is the biggest issue, as the addition of resources and skills to a skillset make the Kripke structure grow exponentially, thus increasing model-checking complexity. This is especially difficult to handle when in the iterative phase of the model definition. Petri net reduction algorithms may be considered as future work if this issue remains or makes the tool inoperable in a realistic development process of an autonomous system.

References

[1] Alexandre Albore, David Doose, Christophe Grand, Charles Lesire & Augustin Manecy (2021): *Skill-Based Architecture Development for Online Mission Reconfiguration and Failure Management*. In: *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, IEEE, pp. 47–54, doi:10.1109/RoSE52553.2021.00015.

[2] Benjamin J. Ayton, Marlyse Reeves, Eric Timmons, Brian C. Williams & Michel D. Ingham (2020): *Toward Information-Driven and Risk-Bounded Autonomy for Adaptive Science and Exploration*. American Institute of Aeronautics and Astronautics, doi:10.2514/6.2020-4149.

[3] Gianfranco Balbo (2001): *Introduction to Stochastic Petri Nets*, pp. 84–155. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/3-540-44667-2_3.

[4] B. Berthomieu, P.-O. Ribet & F. Vernadat (2004): *The tool TINA – Construction of abstract state spaces for petri nets and time petri nets*. *International Journal of Production Research* 42, pp. 2741–2756, doi:10.1080/00207540412331312688.

[5] Bernard Berthomieu, François Vernadat & Silvano dal Zilio (2004): *The TINA toolbox Home Page - Time Petri Net Analyzer - by LAAS/CNRS*. Available at `https://projects.laas.fr/tina/home.php`.

[6] Hugo Costelha & Pedro Lima (2007): *Modelling, analysis and execution of robotic tasks using petri nets*. In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, pp. 1449–1454, doi:10.1109/IROS.2007.4399365.

[7] Hugo Costelha & Pedro Lima (2012): *Robot task plan representation by Petri nets: modelling, identification, analysis and execution*. *Autonomous Robots* 33, pp. 337–360, doi:10.1007/s10514-012-9288-x.

[8] John W. Evans, Frank J. Groen, Lui Wang, Rebekah Austin, Arthur Witulski, Steven L. Cornford, Martin Feather & Nancy Lindsey (2017): *Towards a Framework for Reliability and Safety Analysis of Complex Space Missions*. In: *19th AIAA Non-Deterministic Approaches Conference*, American Institute of Aeronautics and Astronautics, p. 1099, doi:10.2514/6.2017-1099.

[9] Maksym Figat & Cezary Zieliński (2022): *Parameterised robotic system meta-model expressed by Hierarchical Petri nets*. *Robotics and Autonomous Systems* 150, p. 103987, doi:10.1016/j.robot.2021.103987.

[10] Mohammed Foughali, Silvano Dal Zilio & Félix Ingrand (2019): *On the Semantics of the GenoM3 Framework*. Available at `https://hal.laas.fr/hal-01992470`.

[11] Kerianne H. Gross (2017): *Formal specification and analysis approaches for spacecraft attitude control requirements*. In: *2017 IEEE Aerospace Conference*, IEEE, pp. 1–11, doi:10.1109/AERO.2017.7943573.

[12] Adrien Hereau, Karen Godary-Dejean, Jeremie Guiochet & Didier Crestani (2021): *A Fault Tolerant Control Architecture Based on Fault Trees for an Underwater Robot Executing Transect Missions*. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 2127–2133, doi:10.1109/ICRA48506.2021.9561735.

[13] Michel Ingham, Robert Ragno & Brian Williams (2001): *A Reactive Model-based Programming Language for Robotic Space Explorers*. *Proceedings of ISAIRAS-01*. Available at `http://www.ai.mit.edu/projects/ddamba/publications/RMPL.pdf`.

[14] Michel D. Ingham, Robert D. Rasmussen, Matthew B. Bennett & Alex C. Moncada (2006): *Generating requirements for complex embedded systems using State Analysis*. *Acta Astronautica* 58, pp. 648–661, doi:10.1016/j.actaastro.2006.01.005.

[15] Felix Ingrand (2019): *Recent Trends in Formal Validation and Verification of Autonomous Robots Software*. In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*, IEEE, pp. 321–328, doi:10.1109/IRC.2019.00059.

[16] Félix Ingrand & Malik Ghallab (2017): *Deliberation for autonomous robots: A survey*. *Artificial Intelligence* 247, pp. 10–44, doi:10.1016/j.artint.2014.11.003.

[17] Oh Byung Kwon (2004): *Modeling and generating context-aware agent-based applications with amended colored Petri nets*. *Expert Systems with Applications* 27, pp. 609–621, doi:10.1016/j.eswa.2004.06.008.

[18] Charles Lesire & Franck Pommereau (2018): *ASPiC: An Acting System Based on Skill Petri Net Composition*. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 6952–6958, doi:10.1109/IROS.2018.8594328.

[19] Zhifeng Liu & Zhihu Xing (2012): *Characterizing Petri Nets with the Temporal Logic CTL*. In: *Proceedings of 2012 National Conference on Information Technology and Computer Science*, Atlantis Press, pp. 372–375, doi:10.2991/citcs.2012.97.

[20] Silvain Louis, Karen Godary-Dejean, Lionel Lapierre, Thomas Claverie & Sébastien Villéger (2017): *Formal Method for Mission Controller Generation of a Mobile Robot*, doi:10.1007/978-3-319-64107-2_48.

[21] T. Lozano-Perez (1983): *Robot programming*. *Proceedings of the IEEE* 71, pp. 821–841, doi:10.1109/PROC.1983.12681.

[22] Cristian Mahulea & Marius Kloetzer (2018): *Robot Planning Based on Boolean Specifications Using Petri Net Models*. *IEEE Transactions on Automatic Control* 63, pp. 2218–2225, doi:10.1109/TAC.2017.2760249.

[23] Conor Mcgann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn & Rob Mcewen (2008): *T-REX: A Model-Based Architecture for AUV Control*. *3rd Workshop on Planning and Plan Execution for Real-World Systems*. Available at `https://www.researchgate.net/publication/242657580_T-REX_A_model-based_architecture_for_AUV_control`.

[24] Catharine L. R. McGhan, Richard M. Murray, Romain Serra, Michel D. Ingham, Masahiro Ono, Tara Estlin & Brian C. Williams (2015): *A risk-aware architecture for resilient spacecraft operations*. In: *2015 IEEE Aerospace Conference*, IEEE, pp. 1–15, doi:10.1109/AERO.2015.7119035.

[25] I. Mura & A. Bondavalli (2001): *Markov regenerative stochastic petri nets to model and evaluate phased mission systems dependability*. *IEEE Transactions on Computers* 50, pp. 1337–1351, doi:10.1109/TC.2001.970572.

[26] T. Murata (1989): *Petri nets: Properties, analysis and applications*. *Proceedings of the IEEE* 77, pp. 541–580, doi:10.1109/5.24143.

[27] Vittoria Nardone, Antonella Santone, Massimo Tipaldi, Davide Liuzza & Luigi Glielmo (2019): *Model Checking Techniques Applied to Satellite Operational Mode Management*. *IEEE Systems Journal* 13, pp. 1018–1029, doi:10.1109/JSYST.2018.2793665.

[28] Narcis Palomeras, Pere Ridao, Carlos Silvestre & Andres El-fakdi (2010): *Multiple vehicles mission coordination using Petri nets*. In: *2010 IEEE International Conference on Robotics and Automation*, IEEE, pp. 3531–3536, doi:10.1109/ROBOT.2010.5509552.

[29] Sunandita Patra, James Mason, Malik Ghallab, Dana Nau & Paolo Traverso (2021): *Deliberative acting, planning and learning with hierarchical operational models*. *Artificial Intelligence* 299, p. 103523, doi:10.1016/j.artint.2021.103523.

[30] Mikkel Rath Pedersen, Lazaros Nalpantidis, Rasmus Skovgaard Andersen, Casper Schou, Simon Bøgh, Volker Krüger & Ole Madsen (2016): *Robot skills for manufacturing: From concept to industrial deployment*. *Robotics and Computer-Integrated Manufacturing* 37, pp. 282–291, doi:10.1016/j.rcim.2015.04.002.

[31] Enrique Fernández Perdomo, Jorge Cabrera Gámez, Antonio Carlos Domínguez Brito & Daniel Hernández Sosa (2010): *Mission specification in underwater robotics*. *Journal of Physical Agents (JoPha)* 4, pp. 25–33, doi:10.14198/JoPha.2010.4.1.05.

[32] James L. Peterson (1977): *Petri Nets*. *ACM Computing Surveys* 9, pp. 223–252, doi:10.1145/356698.356702.

[33] Christian Reuter, Stefan Göbel & Ralf Steinmetz (2015): *Detecting structural errors in scene-based Multiplayer Games using automatically generated Petri Nets*. *Foundations of Digital Games, Pacific Grove, USA*.

[34] Hassan Reza, Malvika Pimple, Varun Krishna & Jared Hildle (2009): *A Safety Analysis Method Using Fault Tree Analysis and Petri Nets*. In: *2009 Sixth International Conference on Information Technology: New Generations*, IEEE, pp. 1089–1094, doi:10.1109/ITNG.2009.183.

[35] Casper Schou, Rasmus Skovgaard Andersen, Dimitrios Chrysostomou, Simon Bøgh & Ole Madsen (2018): *Skill-based instruction of collaborative robots in industrial settings*. Robotics and Computer-Integrated Manufacturing 53, pp. 72–80, doi:10.1016/j.rcim.2018.03.008.

[36] Franz Steinmetz & Roman Weitschat (2016): *Skill parametrization approaches and skill architecture for human-robot interaction*. In: 2016 IEEE International Conference on Automation Science and Engineering (CASE), IEEE, pp. 280–285, doi:10.1109/COASE.2016.7743419.

[37] Rundong Yan, Lisa M. Jackson & Sarah J. Dunnett (2017): *Automated guided vehicle mission reliability modelling using a combined fault tree and Petri net approach*. The International Journal of Advanced Manufacturing Technology 92, pp. 1825–1837, doi:10.1007/s00170-017-0175-7.

[38] Enrica Zereik, Marco Bibuli, Nikola Mišković, Pere Ridao & António Pascoal (2018): *Challenges and future trends in marine robotics*. Annual Reviews in Control 46, pp. 350–368, doi:10.1016/j.arcontrol.2018.10.002.