ASPiC: an Acting system based on Skill Petri net Composition

Charles Lesire[1] and Franck Pommereau[2]

Abstract— **While developing automated planning algorithms helps in making robots more autonomous, the development of acting systems is also of major concerns. Acting systems aim at refining high-level actions into executable commands, while managing access to resources, possible failures, or any other kind of unpredictable situation from the planner point of view. Improving the trust on autonomous robots also requires to have a formal model of acting, and the capability to perform some analysis on this model. In this paper, we present ASPiC, an acting system based on the modeling of robot's skills using a specific control-flow Petri net model. The skills can then be combined using well-defined operators to build a complete plan that refines a high-level action. Some good properties are guaranteed by construction, while others can be verified on the resulting plan model. This paper also presents the application of ASPiC to an area protection mission by an autonomous surface vehicle.**

I. Introduction

Autonomous robotics researches bubble up. Recent works on making robots autonomous integrate advanced AI techniques, such as learning, decision theory, or automated planning. These techniques allow the robots to reason about their states and goals in order to choose the appropriate actions to execute.

However, few works deal with the way to execute correctly these actions. Managing action execution calls for refining symbolic actions into executable commands, monitoring their execution, and reacting to failures and hazards. All these features are brought together under the term *Acting* [1], [2]. While acting systems were an active field a couple of decades ago (especially for space exploration systems [3]), they have been a bit abandoned in favor of pure AI. Nevertheless, autonomous robots make it necessary to demonstrate the safety and reliability of autonomous acting behaviors.

In this paper, we present ASPiC, an acting system that is based on the Petri net formalism. Petri nets are a natural choice to model concurrent systems as they allow to precisely represent the management of shared resources among concurrent processes, with a natural expression of conflicts, independence, causality, etc. Moreover, Petri nets form a very flexible family of formalisms which is easy to adapt to our exact needs, and they come with a wide range of analysis techniques as well as numerous tools already available. In particular, model-checking allows to formally assess dynamic properties of Petri net models in an automated way. ASPiC is based on composition of elementary Petri nets, representing the basic skills of the system. Using such compositions allows first to build an executable model of a plan in a hierarchical way, and second to ensure some sound construction of the acting model.

Section II presents some works related to acting in robotics. Petri net skill models and their composition to build an action plan is presented in Sect. III. The implementation of ASPiC and its application to a marine area protection is presented in Sect. IV.

II. Related Works

According to [1], *Acting* is the component of a deliberative architecture that has to manage: plan refinement (*i.e.*, how to decompose a plan action down to executable commands), reaction to events (that may require plan adaptation or plan refinement adaptation), time management (when planned actions consider deadlines, durations, or time constraints), non-determinism (*i.e.*, partial or noisy observations during the mission execution), and plan repair (when, why, how to trigger the planning component).

Historical researches on acting systems have essentially dealt with *execution control*, *i.e.*, the way to refine plans into commands. Several execution frameworks used in space robotics are described in [3]. Their analysis has led to Plexil [4], a language and an executive that aggregates the features of a lot of former execution control systems. Plexil provides constructs to describe hierarchical decomposition of plans into a tree of nodes. Leaf nodes are commands sent to the physical system or read-access to system variables. The Plexil language allows to describe sequences of nodes, concurrent execution, branches and loops. However, it does not rely on a formal mathematical model, and does not support advanced features to change the refinement according to events, nor provides analysis tools.

Among the architectures for autonomy that integrate planning and acting, T-REX [5] made a step towards the formalization of the refinement process, by specifying exchanges between *reactors* through the manipulation of timelines. The hierarchical (and temporal) decomposition of the decision is then clearly specified through access to timelines (one timeline is writable only by one reactor). However, the implementation of the reactors is not formalized, then preventing to make some safety analysis of the overall system.

Safety and reliability analysis in acting systems requires formal models of acting. The ROS ecosystem provides some ways of specifying action refinements, through hierarchical state-machines [6], [7]. In these tools, the basic elements correspond to ROS primitives (*e.g.*, topic listening or service calls), and the design process is very permissive. Even if some analysis may be possible on these models, it would

[1]Charles Lesire is with ONERA – The French Aerospace Lab, Toulouse, France. charles.lesire@onera.fr

[2] IBISC laboratory, university of Évry / Paris-Saclay, Évry, France. franck.pommereau@univ-evry.fr

be interesting to restrain the possible constructs to define some patterns that preserve some properties by construction. Moreover, basic concepts are ROS-specific, that would make difficult to reuse the models and analyses in another context.

Other works used Petri nets as a formal model of robot actions. [8] models elementary actions by generalized stochastic Petri nets, with one place representing the action instance, and one place for each predicate used as a precondition or an effect of the action. The Petri net model corresponds to a PDDL-like description of actions, and predicate places are then used to causally link actions, as a task planning algorithm would do. These Petri net models are not made to describe operational behaviors: there is no specification of how to execute commands, nor composition operators allowing to describe branches, loops, or other operational constructs. Petri Net Plans [9] formalize elementary actions in term of phases: an *initial* place, an *execution* place, and a *termination* place. They distinguish ordinary (deterministic) actions and sensing (Boolean) actions. They also define operators to combine actions: sequences, conditions, iterations and interruptions. However, they do not discuss formal properties or analysis of the resulting Petri net plan, and they do not manage failures as a possible action outcome.

While [9] makes an ad-hoc description of elementary actions, some common Petri net frameworks exist to model such processes, for instance workflow nets [10] (to model business processes and operations), and M-nets [11]. In this paper we use an algebra of colored Petri nets inspired from [12], [13]: on the one hand they form a variant of co-loured Petri nets [14] whose tokens carry values on which arbitrary computation can be performed; on the other hand they have an explicit control-flow that enables for composition operations like it is usual with process algebras. We borrow the classical operators of sequential composition, choice, and parallel composition as found in [13], and we adapt them to handle errors (exceptions) as in [12] but in a simpler way that is more suited to our context. We also define new operators that were not considered in the past.

III. ASPiC

ASPiC is an acting system based on Petri net compositions. Compositions allow to assemble elementary actions into more complex plans through consistent operators. Elementary actions are often called *skills* [2], as they correspond to the several capacities or behaviors that are available at the platform level. For instance, *skills* was the term used for the elementary behaviors of the first behavior-based architectures for autonomous navigation [15], [16].

ASPiC formalizes skills through specific models of Petri nets using control-flow semantics. These skills can then be composed to form an action plan, through operators that manage nominal execution (sequences, concurrence), observations (through conditions evaluated at execution time), or failures (using exception management).

A. Background and basic definitions

To start with, let us recall that a multiset is a collection of unordered elements allowing repetitions. For instance $\{a, a, b\}$ is the multiset with two a's, one b and no other values. Multisets may be added, subtracted, compared, or multiplied by non negative integers. For instance, we have $\{a, b\} + \{a\} - \{b\} = 2 \times \{a\} \leq \{a, a, a, b\}$. Curly brackets around multisets my be omitted for brevity. We note by $X^\star$ the set of multisets over X.

We consider a variant of Petri nets colored by an abstract *color domain*, which is both more general than a specific programming language, and simpler to define. We note by $\mathbb{D}$ the set of all the data values (*e.g.*, integers, Boolean values True and False, regular "black token" $\bullet$, "white token" $\circ$ for errors, etc., including data structures) and assume that there is a value $\bot \notin \mathbb{D}$ corresponding to the undefined value. We note by $\mathbb{E}$ the set of all the expressions, built on the top of $\mathbb{D}$ and a set $\mathbb{V}$ of variables. Given $e \in \mathbb{E}$, we note by $\mathrm{vars}(e) \subseteq \mathbb{V}$ the set of variables involved in e. We may evaluate e using a *binding* β that is a function $\mathrm{vars}(e) \to \mathbb{D} \cup \{\bot\}$, and we note by $\beta(e)$ the result of evaluating e with respect to β; this may be a valid value if the evaluation is possible, or $\bot$ if anything goes wrong during the evaluation (*e.g.*, a syntax or typing error, a division by zero, etc.). For instance, for $\beta \stackrel{\text{df}}{=} \{x \mapsto 1, y \mapsto 2\}$ we usually have $\beta(x + y) = 3$ and $\beta(y/(x - 1)) = \bot$ (which actually depends on the concrete col-our domain considered).

We now define our variant of colored Petri nets with control-flow, that is inspired from [12], [13] and adapted to our specific needs in this paper.

Definition 1 (control-flow Petri nets): A *control-flow Petri net* (*CFPN*) is a tuple $N \stackrel{\text{df}}{=} (P, T, \ell, \sigma)$ such that:

- P is the non-empty set of *places*;
- T, disjoint from P, is the non-empty set of *transitions*;
- ℓ is the *labeling* function such that:
  - for all $p \in P$, $\ell(p) \subseteq \mathbb{D}$ is the *type* of p, *i.e.*, the data values it may contains,
  - for all $t \in T$, $\ell(t) \in \mathbb{E}$ is the *guard* of t, *i.e.*, a Boolean function that serves as a condition for the firing of t,
  - for all $(x, y) \in (P \times T) \cup (T \times P)$, $\ell(x, y) \in \mathbb{E}^\star$ is the *arc* from x to y, *i.e.*, the multiset of expressions representing the tokens carried by the arc;
- σ is the *status* function such that:
  - for all $p \in P$, $\sigma(p) \in \{\mathsf{e}, \mathsf{i}, \mathsf{x}, \varepsilon\} \uplus \mathbb{B}$ where e denotes an *entry* place, i denotes an *internal* place, x denotes an *exit* place, all together forming the control-flow places, ε denotes a *buffer* place that is not shared (private to the net), and any status in $\mathbb{B}$ denotes a *buffer* places to be shared with other nets,
  - for all $t \in T$, $\sigma(t) \in \mathbb{N} \uplus \{\bot\}$ denotes whether t is a regular transition ($\bot$) or a placeholder transition (any $n \in \mathbb{N}$) intended to be substituted by the nth argument of a net operation (see Sect. III-D). We assume that $\{\sigma(t) \mid t \in T\} \setminus \{\bot\}$ is an initial segment of $\mathbb{N}$, *i.e.*, that placeholder transitions are

numbered starting from zero without any gap,
- for all $(x, y) \in (P \times T) \cup (T \times P)$, $\sigma(x, y)$ is a function $\mathbb{D}^\star \times \mathbb{D}^\star \to (\mathbb{D} \cup \{\bot\})^\star$ that, given the marking of a place and the evaluation of an arc annotation, returns the actual multiset of consumed or produced tokens. We shall use in particular, for a regular arc:

$$\sigma_= \stackrel{\text{df}}{=} (m, a \mapsto a) \tag{1}$$

and, for an inhibitor arc:

$$\sigma_\neg \stackrel{\text{df}}{=} (m, a \mapsto \emptyset \text{ if } m - a = m \text{ else } \{\bot\}) \tag{2}$$

$\Diamond$

We also adopt the following notations:

$$P^\circ \stackrel{\text{df}}{=} \{p \in P \mid \sigma(p) \in \{\mathsf{e}, \mathsf{i}, \mathsf{x}\}\} \tag{3}$$

$$\forall t \in T,\ {}^\bullet t \stackrel{\text{df}}{=} \{p \in P \mid \ell(p, t) \neq \emptyset\} \tag{4}$$

$$\forall t \in T,\ t^\bullet \stackrel{\text{df}}{=} \{p \in P \mid \ell(t, p) \neq \emptyset\} \tag{5}$$

The dynamics of CFPN need not be precisely defined in this paper and we refer the reader to [12], [13] for details. Intuitively, Petri nets are marked by tokens that are multisets of values from the type of each place. Then, given a binding β, the arcs surrounding a transition t can be evaluated through β and σ and t may fire iff (1) there are enough tokens to be consumed as specified by the input arcs (from ${}^\bullet t$ to t), (2) the guard of t evaluates to True, and (3) the tokens produced by the output arcs (from t to $t^\bullet$) are in the type of the output places. When t is fired, it consumes and produces tokens as specified by its arcs (and σ).

In this paper, we will consider a subset of the class of CFPN that respect some syntactical constraints, thus we define well-formedness as follows.

Definition 2 (well-formedness): A CFPN (P, T, ℓ, σ) is called *well-formed* iff:

1) For all $p \in P^\circ$ we have $\ell(p) = \{\bullet, \circ\}$.
2) N has exactly one entry place.
3) N has exactly one exit place.
4) N has at most one place of each status in $\mathbb{B}$. $\Diamond$

Black tokens $\bullet$ are used to represent the regular execution of the control-flow, while white tokens $\circ$ represent errors or exceptions.

B. Skill Petri nets

A *skill* represents a behavior or a capacity of a robotic system. It corresponds to commands or behaviors that can be triggered on the robot platform. These behaviors may not only deal with achievement of movements, but also with observations. Performance Level Profiles [17] is a recent work that has proposed four types of modules available at the robot platform level: *Achieve*, *Observe*, *Maintain* and *Detect*, the two last being special cases or constructs of the two former. The modules definition includes input and output parameters, as well as required resources. A skill, that corresponds to one of these modules, can then be seen as a remote procedure available on the robot platform, that will be called and managed by the Acting system. Although we used the description in [17] as a starting point for describing skills, the models described hereafter are generic enough to meet any similar representation of robots' skills we found in the literature.

A *skill Petri net* (*SkPN*) models such a skill (see an example in Fig. 1) and is parametrized by the *resources* it requires in order to be executed, which may be locks (mutex on resources), inputs (the states it reads upon start-up), or outputs (the states it updates on completion). More precisely, we consider two kind of resources:

- locks from a set $\mathbb{L}$ are resources whose access is exclusive and may be reserved or released, but that are not associated with a particular value. However, several instances of a lock may be available and several instances may be reserved or released at the same time. Locks will be modeled by buffer places marked with as many black-tokens $\bullet$ as the number of instances;
- states from a set $\mathbb{S}$, disjoint from $\mathbb{L}$, are resources associated with a value which may be read (inputs) or updated (outputs). States will me model-led by buffer places as well, whose marking is a single token in $\mathbb{D}$ representing the current value of the state.

To conveniently model resources as places, we assume that each resource is a valid buffer place status, *i.e.*, $\mathbb{L} \uplus \mathbb{S} \subset \mathbb{B}$.

Intuitively, a SkPN is then defined by: its control flow places, a transition t_{start} that triggers the start of the skill execution, a transition t_{stop} fired when the skill nominally ends, a transition t_{except} fired when the skill fails, and a place p_{exec} that stores the skill's state during its execution. These nodes are surrounded with buffer places to model the resources used by the skill: when t_{starts} fires, it reads the skill's inputs and acquires its locks, then when t_{stop} (or t_{except}) fires, it writes the skill's outputs and releases the locks.

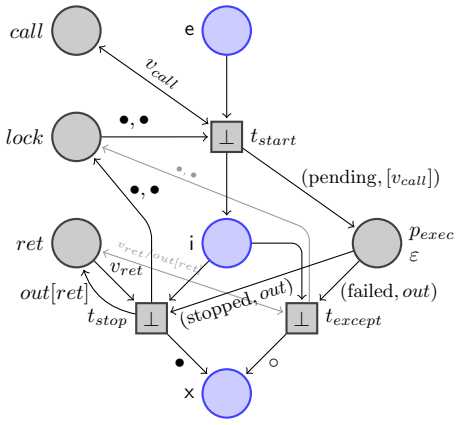


Fig. 1: A Skill Petri net with $\mathcal{I} \stackrel{\text{df}}{=} \{call\}$, $\mathcal{L} \stackrel{\text{df}}{=} \{lock \mapsto 2\}$, and $\mathcal{O} \stackrel{\text{df}}{=} \{ret\}$. Control-flow places are depicted in blue, the other places represent resources. Places' labels are their status except for p_{exec} whose name is also indicated. Some labels $\bullet$ on arcs are omitted as well as curly brackets around multisets. The arcs between t_{except} and $ret/lock$ are the same as for t_{stop} as sketched in gray.

↪	t_{start}	t_{stop}	t_{except}
p_{e}	•		
p_{i}		•	•
p_{x}			
p_{exec}		(stopped, out)	(failed, out)
p_l	$\mathcal{L}(l) \times \bullet$		
p_i	v_i		
p_o		v_o	v_o

↩	t_{start}	t_{stop}	t_{except}
p_{e}			
p_{i}	•		
p_{x}		•	○
p_{exec}	(pending, $[v_i \mid i \in \mathcal{I}]$)		
p_l		$\mathcal{L}(l) \times \bullet$	$\mathcal{L}(l) \times \bullet$
p_i	v_i		
p_o		$out[o]$	$out[o]$

Fig. 2: Specification of the arcs in a SkPN with input arcs on the left-hand table and output arcs on the right-hand table, where p_l is the place that models a lock l, p_i is the place that models an input i, and p_o is the place the model an output o. Note that we may have some i's equal some o's in which case both types of arcs are to be considered. Unspecified arcs are the empty multiset and all status are $\sigma_=$. Curly brackets around multisets (for ℓ) have been omitted.

Definition 3 (skill Petri nets): A *skill Petri net* (*SkPN*) is a well-formed CFPN (P, T, ℓ, σ) parametrized by $\mathcal{L}$, $\mathcal{I}$, and $\mathcal{O}$ such that:

- $\mathcal{L}$ is a function $\{l_1, \ldots, l_n\} \subseteq \mathbb{L} \to \mathbb{N}$ representing the locks required by the SkPN together with the number of instances of each;
- $\mathcal{I}$ is a set $\{i_1, \ldots, i_m\} \subseteq \mathbb{S}$ representing the inputs of the SkPN;
- $\mathcal{O}$ is a set $\{o_1, \ldots, o_k\} \subseteq \mathbb{S}$ representing the outputs of the SkPN;
- its entry, internal, and exit are respectively called p_{e}, p_{i}, and p_{x};
- there is a place $p_{exec} \in P$ such that $\sigma(p_{exec}) = \varepsilon$ and $\ell(p_{exec}) = \mathbb{D} \times ([\mathbb{D}]_{\mathcal{I}} \cup [\mathbb{D}]_{\mathcal{O}})$, where $[\mathbb{D}]_X$ denotes the set of all vectors on $\mathbb{D}$ indexed by X;
- for each $l \in \mathrm{dom}(\mathcal{L})$ there is a place $p_l \in P$ such that $\ell(p_l) = \{\bullet\}$ with $\sigma(p_l) = l \in \mathbb{B}$ is a unique status corresponding to the resource name;
- for each $i \in \mathcal{I}$ there is a place $p_i \in P$ such that $\ell(p_i) = \mathbb{D}$ and $\sigma(p_i) = i$;
- for each $o \in \mathcal{O}$ there is a place $p_o \in P$ such that $\ell(p_o) = \mathbb{D}$ and $\sigma(p_o) = o$;
- $T \stackrel{\text{df}}{=} \{t_{start}, t_{stop}, t_{except}\}$ such that $\ell(t) = \mathrm{True}$ and $\sigma(t) = \perp$ for all $t \in T$;
- arcs as specified in Fig. 2. ◊

C. Execution handlers

Skills are behaviors, or processes, available at the robot platform level. In order to interact with actual skills execution, the Acting system needs to manage *skill handlers*. SkPN must then be equipped with skill handlers in order to interact with skill execution, *i.e.*, start skill execution, monitor success or failures, get returned values, etc.

A skill handler can be defined as a Petri net that contains an execution place p_{exec} that will be merged with the p_{exec} of the SkPN. Figure 3 shows a basic handler used to simulate the execution of skills. An handler to interact with ROS actions is presented in Sect. IV-C.

We thus define a handler net and the operation that plugs it onto a SkPN.

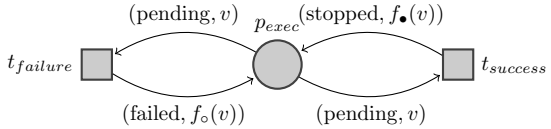


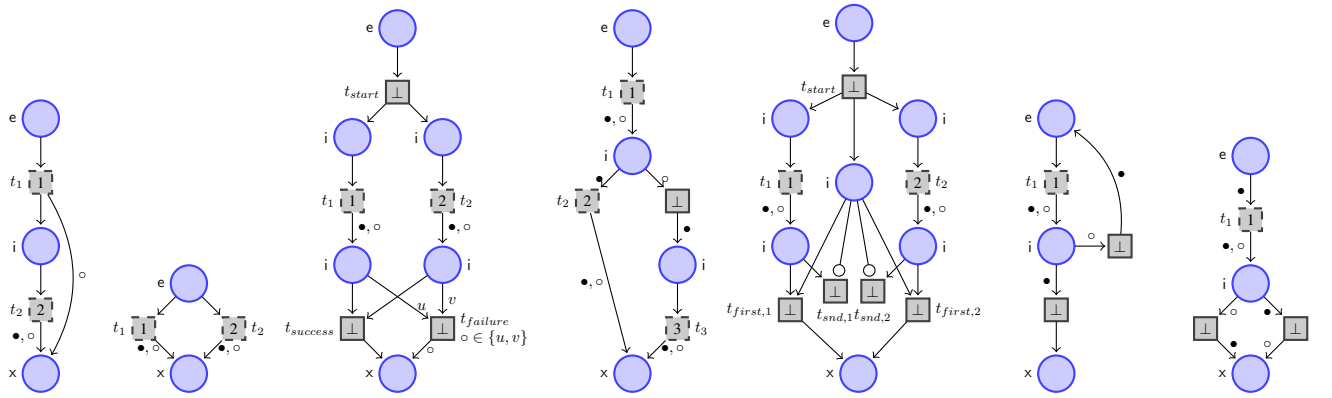
Fig. 3: Basic handler leading to success or failure. This handler is used to simulate or analyze the SkPN behavior. Functions $f_\bullet$ and $f_\circ$ respectively simulate the skill outputs given the inputs v.

Definition 4 (handlers): A *handler* is a CFPN $N_h \stackrel{\text{df}}{=} (P_h, T_h, \ell_h, \sigma_h)$ that has no entry nor exit place, its internal places have type $\{\bullet, \circ\}$, and is such that there is exactly one $p_{exec} \in P_h$ such that $\ell_h(p_{exec}) = \mathbb{D} \times ([\mathbb{D}]_{\mathcal{I}} \cup [\mathbb{D}]_{\mathcal{O}})$ and $\sigma_h(p_{exec}) = \varepsilon$. Let $N = (P, T, \ell, \sigma)$ be a SkPN, we note by $N \boxdot N_h$ the SkPN $(P \cup P_h, T \cup T_h, \ell \cup \ell_h, \sigma \cup \sigma_h)$ in which the place p_{exec} originated from N and that originated from N_h have been merged, as well as all the places sharing the same status in $\mathbb{B}$. ◊

D. ASPiC operators

Operators are aimed at modifying or composing Petri nets from skills up to action plans. An operator is defined by a so-called *operator net* (noted N_0 in the following definition) that is a CFPN with placeholder transitions aimed to be substituted with the operators arguments (so-called *operand nets*). This substitution consists in connecting the control-flow of operand nets as specified in the operator net, then merging the buffer places representing the same resources. Formally we have:

Definition 5 (SkPN operators): Take $n > 0$ and let $\{N_1, \ldots, N_n\}$ be n CFPN such that $N_i \stackrel{\text{df}}{=} (P_i, T_i, \ell_i, \sigma_i)$. Let $N_0 \stackrel{\text{df}}{=} (P_0, T_0, \ell_0, \sigma_0)$ be a CFPN such that it has n placeholder transitions $\{t_1, \ldots, t_n\}$. We note by e_i and x_i the entry and exit place of each N_i for $0 \leq i \leq n$. We assume $P_i \cap P_j = \emptyset = T_i \cap T_j$ for $0 \leq i \neq j \leq n$. This allows to define a n-ary operation noted as $N_0[N_1, \ldots, N_n]$ that substitutes each t_i in N_0 with the corresponding N_i, yielding a new CFPN that is defined in two steps. First, transitions t_i's are substituted by nets N_i's whose control-



(a) $N_{\odot}$ (b) N_{+} (c) $N_{|}$ (d) N_{ite} (e) $N_{\otimes}$ (f) N_{retry} (g) $N_{\neg}$

Fig. 4: ASPiC operator nets. All places are control-flow places whose status is depicted as a label. Dashed transitions are placeholders, transitions status are depicted inside the transitions, and omitted guards are True. Omitted arcs labels are • and all arc status are $\sigma_{=}$ except inhibitor arcs ($\multimap$ shaped) whose status is $\sigma_{\neg}$.

flows are connected, which builds an intermediary CFPN $N \stackrel{\text{df}}{=} (P, T, \ell, \sigma)$ defined by:

- $P = P_0 \cup \bigcup_{1 \leq i \leq n} (P_i \setminus \{e_i, x_i\})$;
- $T = (T_0 \setminus \{t_1, \ldots, t_n\}) \cup \bigcup_{0 \leq i \leq n} T_i$;
- $\ell = \ell' \cup \bigcup_{0 \leq i \leq n} \sigma_i|_{P,T}$ and $\sigma = \sigma' \cup \bigcup_{0 \leq i \leq n} \sigma_i|_{P,T}$, where $\ell_i|_{P,T}$ (resp. $\sigma_i|_{P,T}$) is ℓ_i (resp. σ_i) restricted to the domain $(P_i \cap P) \cup (T_i \cap T) \cup ((P_i \cap P) \times (T_i \cap T))$, and with ℓ' and σ' defined as the smallest functions such that:
  - for every $1 \leq i \leq n$, every $c \in P_0^{\bullet\circ}$, and every $t \in T_i$ such that $\ell_i(e_i, t) \leq \ell_0(c, t_i)$, then we have $\ell'(c, t) = \ell_i(e_i, t)$ and $\sigma'(c, t) = \sigma_0(c, t_i)$,
  - for every $1 \leq i \leq n$, every $c \in P_0^{\bullet\circ}$, and every $t \in T_i$ such that $\ell_i(t, x_i) \leq \ell_0(t_i, c)$, then we have $\ell'(t, c) = \ell_i(t_i, x_i)$ and $\sigma'(t, c) = \sigma_0(t_i, c)$.

Then, the result of $N_0[N_1, \ldots, N_n]$ is defined from N as follows: for every $\mathsf{stat} \in \mathbb{B}$ we merge all the places in $\{p \in P_i \mid 0 \leq i \leq n, \sigma_i(p) = \mathsf{stat}\}$. ◊

We consider specifically the operators nets depicted in Fig. 4 and we use natural notations for the operations, for instance $N_1 \odot N_2$ instead of $N_{\odot}[N_1, N_2]$, $retry(N)$ instead of $N_{retry}[N]$, or $\neg N$ instead of $N_{\neg}[N]$.

1) Sequential composition: $N_1 \odot N_2$ (Fig. 4a) enforces the execution of N_1 followed by that of N_2, except if N_1 terminates with an error, putting ○ in its exit place, in which case the execution of N_2 is skipped thanks to the arc on the right side.

2) Choice: $N_1 + N_2$ (Fig. 4b) allows to execute either N_1 or N_2, which is chosen in a non-deterministic way if both are possible.

3) Concurrent composition: $N_1|N_2$ (Fig. 4c) allows to execute both N_1 and N_2 concurrently. When both are terminated, the whole composition results in a success if both nets succeeded (through transition $t_{success}$), or a failure if at least one of the two nets has failed (through transition $t_{failure}$).

4) If-Then-Else: A common situation that arises when decomposing actions is to react to the successful or failed execution of a skill. In that case, one may want to execute the sequel of a nominal plan in case of success, and perform a specific action in case of failure. $ite(N_1, N_2, N_3)$ (Fig. 4d) allows to handle such a situation by executing first N_1, and then N_2 if N_1 succeeds, otherwise it executes N_3.

5) Race composition: The concurrent composition $N_1|N_2$ can only terminate when both N_1 and N_2 have terminated. In some situations, one may want to execute N_1 and N_2 concurrently, and continue the plan execution as soon as one of both has finished. $N_1 \otimes N_2$ (Fig. 4e) allows this by starting both N_1 and N_2 but then, if N_1 terminates first, it allows the firing of $t_{first,1}$ which terminates the whole composition; then, when N_2 terminates, its resulting token is cleared thanks to transition $t_{snd,2}$. The situation is symmetrical if N_2 terminates first. Selecting between $t_{first,i}$ and $t_{snd,i}$ is ensured through the central internal place and the inhibitor arcs: when the place is marked (after t_{start}), only the $t_{first,i}$'s are possible and their firing consumes the token, which allows then only the $t_{snd,i}$'s.

6) Retry: $retry(N)$ (Fig. 4f) is another common construct related to exception management. It allows to retry the execution of N in case of error, until an execution eventually succeeds.

7) Negation: $\neg N$ (Fig. 4g) is the negation operator that inverts the termination status of N by transforming a regular token into an exception token and vice-versa. It is useful when a skill returns successfully but from a supervision point of view, this success must be considered as a failure, for instance for skills that detect abnormal situations.

E. Initial marking

When a system has been fully composed, the resulting Petri net has to be marked before it is executed.

Definition 6 (initial marking): Let $N \stackrel{\text{df}}{=} (S, T, \ell, \sigma)$ be a well-formed CFPN. An *initial marking* of N is a marking M such that:

- $M(p_{\mathsf{e}}) = \{\bullet\}$ where p_{e} is the entry place;

- for all $l \in \mathbb{L}$ such that we have $p_l \in P$ with $\sigma(p_l) = l$, then $M(p_l) = k_l \times \{\bullet\}$ where $k_l \geq 0$ is the number of available instances of resources l in the system modeled;
- for all $s \in \mathbb{S}$ such that we have $p_s \in P$ with $\sigma(p_s) = s$, then $M(p_s) = \{v_s\}$ where $v_s \in \mathbb{D}$ is the initial value of state s in the system modeled;
- no other place is marked. ◊

F. Properties

Specifying an acting system using Petri nets allows to perform some analysis to verify that the plan refinement under execution has some good properties. By enforcing a structured way of modeling systems, ASPiC also aims at providing some of these properties *by construction*.

1) Well-formedness: This property ensures that our operations are well defined (in particular, we have the required entry/exit places to correctly apply the substitutions of placeholders transitions). First, we state that the result of plugin a handler onto a SkPN is well formed; then we state that our operations preserve well-formedness.

Proposition 1: Let N_h be a handler and N a SkPN, then $N \circledcirc N_h$ is well-formed. □

Proof: Condition (1) holds because N is well-formed and the internal places of N_h respect the condition. Conditions (2) and (3) hold because N respect them and N_h has no entry nor exit place. Condition (4) holds because operation $\circledcirc$ explicitly merges the places with the same status. ■

Proposition 2: Let N_0 be one of the operator nets from Fig. 4, let n be the number of placeholder transitions in N, and let $N_1, \ldots, N_n$ be n well-formed nets. Then, $N \stackrel{\text{df}}{=} N_0[N_1, \ldots, N_n]$ is well-formed. □

Proof: Condition (1) holds on all the N_i's for $0 \leq i \leq n$ so it also holds on N whose places are all copied from the N_i's. Conditions (2) and (3) hold because the composition removes the entry and exit places from the operand nets and preserves only those from N that itself respects the condition. Condition (3) holds because the operation explicitly merges the places with the same status. ■

Well-formedness is also desirable because it ensures a consistent handling of resources through condition (4): there is only one place for each resource so that locks and states values are uniquely represented. In particular, acquiring/releasing a lock is global on the system, and every state has a unique value across all the system.

2) Soundness: This property can be considered as the behavioral version of well-formedness. It guarantees that only "good" executions can occur, *i.e.*, that the Petri net is well-behaved. This can be verified on the state space or guaranteed by construction through syntactical restrictions like those we have considered for well-formedness.

Various definitions of soundness have been considered in the literature, depending on the particular needs of the considered applications. For instance, in workflow PN [10], soundness is defined by:

$$\forall M, ([p_{\mathsf{e}}] \stackrel{*}{\rightarrow} M) \Rightarrow (M \stackrel{*}{\rightarrow} [p_{\mathsf{x}}]) \tag{6}$$

$$\forall M, ([p_{\mathsf{e}}] \stackrel{*}{\rightarrow} M \wedge M \geq [p_{\mathsf{x}}]) \Rightarrow (M = [p_{\mathsf{x}}]) \tag{7}$$

$$\forall t, \exists M, M', [p_{\mathsf{e}}] \stackrel{*}{\rightarrow} M' \stackrel{t}{\rightarrow} M \tag{8}$$

This can be read as: (6) any reachable marking allows to reach an exit marking (*i.e.*, a marking in which the exit place is marked), (7) in such a case, the exit place is the only marked place, and (8) every transition may be fired following a path from the initial marking. Properties (6) and (8) make sense for workflows but are rarely found in general because they depend too much on the system being modeled. For instance, (6) may be relaxed to requiring that the only deadlocks are exit markings, but the system may never reach them (which is usually highly desired for a controller that is not supposed to stop). In [18], property (7) is named cleanness and has been relaxed in [19] when buffer places have been introduced. Indeed, such places may retain tokens from one execution of a subnet to be used in a further execution. This is the case in ASPiC where buffer places model the resources of the system which are globally available. The solution in [19] was to restrict cleanness to the control-flow places, which can be formulated here for any initial marking M_0 such that $M_0(p_{\mathsf{e}}) \in \{\{\bullet\}, \{\circ\}\}$ and $p \in P^{\bullet} \setminus \{p_{\mathsf{e}}\} \Rightarrow M_0(p) = \emptyset$ as:

$$\begin{array}{rl} \forall M: & M_0 \stackrel{*}{\rightarrow} M \wedge M(p_{\mathsf{x}}) \neq \emptyset \\ \Rightarrow & M(p_{\mathsf{x}}) \in \{\{\bullet\}, \{\circ\}\} \\ & \wedge (p \in P^{\bullet} \setminus \{p_{\mathsf{x}}\} \Rightarrow M(p) = \emptyset) \end{array} \tag{9}$$

In the current state of ASPiC, this property can only be verified dynamically because operator $\otimes$ is not clean and allows to reach an exit marking while one of the nets is still active (and thus has its control-flow marked).

[18], [19] also introduce another notion of well-behavior through control-safety that states that a control-flow place may not be marked by more than one token. This is desirable to ensure that at any point of a system execution, each activity cannot be activated more than once, in particular, a skill net cannot be multiply activated which would lead to several simultaneous invocations of the underlying skill. Like cleanness, control-safety is guaranteed by constructions in [18], [19] thanks to syntactic restrictions on the basic (here SkPN) and operator nets.

Unfortunately, $\otimes$ is not control-safe when nested into a $retry$: a new instance of the still active net may be started when the $retry$ loops.

We thus need to find a better definition for $N_{\otimes}$ before to be able to provide a satisfactory notion of soundness that would encompass cleanness and control-safety and would be guaranteed by construction. This is left as a future work. In the meantime, this property must be checked by computing the state space of the Petri net. However, if $\otimes$ is not used, it is likely that we have such a property because our operator nets and the SkPN basically respect the syntactical restrictions used in [18], [19] to enforce cleanness and control-safety.

IV. Area Protection with an Autonomous Surface Vehicle

A. Mission description

We consider an Autonomous Surface Vehicle (see Fig. 5) that performs an area protection mission. That ASV has to patrol within a given zone and detect intruders. On detection, the ASV may have to intercept the intruder. The ASV also avoids obstacles on its path. After an avoidance or an interception, the patrolling is resumed.



Fig. 5: An Autonomous Surface Vehicle controlled by ASPiC

B. ASV Skills

The ASV platform provides several skills, defined following the concepts and types described in [17], each skill being managed by a SkPN:

- *Achieve* skills:
  - N_{traj}: execute a given trajectory;
  - N_{goto}: move the ASV to a target pose;
  - $N_{intercept}$: execute the interception maneuver;
- *Observe* skills:
  - N_{comp}: compute a patrolling trajectory according to a given start pose, and a patrolling duration;
  - N_{pose}: observe the current pose of the ASV;
- *Maintain* skills:
  - N_{keep}: maintain the ASV at its current pose;
- *Detect* skills:
  - N_{obs}: detect an obstacle triggering an avoidance;
  - N_{avoid}: detect the end of an avoidance maneuver (the avoidance itself is automatically executed by the platform).

These skills are implemented using the MAUVE middleware [20] and provide a ROS *actionlib* interface in order to be manageable by ASPiC.

C. ASPiC Implementation

ASPiC has been implemented as a ROS node that receives actions to perform from a Planning node. SkPN and composition operators have been implemented in Python, using the SNAKES library [21]. To interface with the ASV platform, new handlers have been defined to manage ROS actions. The ROS handler (see Fig. 6) has a behavior that abstracts the state-machine of the ROS actionlib client. The p_{cl} place owns a ROS actionlib client object cl. When the status is $\mathrm{pending}$, transition $t_{activate}$ can fire, and calls the ROS action server

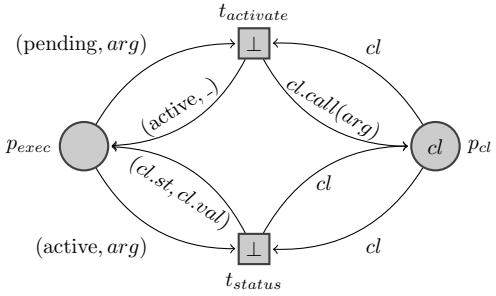


Fig. 6: ROS actionlib handler N_{ros}

to execute the action. The skill status is then active. Then each time transition t_{status} is fired, the status is updated from the client object (using $cl.st$) and in case of termination (either succeeded or failed), the return value is retrieved using $cl.val$.

D. Specification of ASV actions

As discussed in the introduction, ASPiC is aimed to be the acting system of a deliberative architecture. It is then supposed to be fed up with action plans coming from a Planning component.

In the ASV mission, we consider that the Planning component plans *patrol* actions of a given duration, *intercept* actions when intruders must be intercepted, and *station* actions in which case the ASV has to join a position and keep it. ASPiC has then to refine these actions and monitor their execution. The *intercept* actions are directly mapped to the execution of the interception skill through the SkPN $N_{intercept}$. The *station* actions have simple decompositions combining a movement to the position to hold, and then keeping this position, as described by Eq. (10), where $N_i^{\boxdot}$ denotes SkPN N_i combine with a ROS handler.

$$N_{goto}^{\boxdot} \odot N_{keep}^{\boxdot} \tag{10}$$

The *patrol* actions are more interesting as ASPiC uses a more complex decomposition, including obstacle avoidance monitoring. The expression defining the *patrol* action is given by Eq. (11).

$$\begin{aligned} retry(ite(&N_{pose}^{\boxdot} \odot N_{comp}^{\boxdot} \odot (N_{traj}^{\boxdot} \otimes (\neg N_{obs}^{\boxdot})),\\ &N_{pose}^{\boxdot} \odot N_{keep}^{\boxdot},\\ &\neg N_{avoid}^{\boxdot})) \end{aligned} \tag{11}$$

The first part of the action decomposition (first argument of *ite*) is a sequence of observing the current (*i.e.*, initial) pose, computing a patrol trajectory, and concurrently executing this trajectory while detecting obstacles. In case of success (trajectory completely followed), the second part of the *ite* composition is executed. It consists of a sequence of observing the current (*i.e.*, final) pose, and maintain this pose. In case of failure (*i.e.*, an obstacle triggered an automatic avoidance, or the trajectory failed for another reason), the last part of *ite* is executed: ASPiC waits for the detection of

the avoidance end, then the *retry* operator will lead to a new patrol computation from current pose.

E. Experiments and Results

Several area protection missions have been realized with the ASV, on several areas, including obstacles and target interceptions. In total, the ASV has been under ASPiC control for 5422 seconds. In these missions, ASPiC has managed in total 17 *patrol* actions, 24 *intercept* actions, 7 *station* actions, and 14 obstacle avoidances that led to retry the patrol action. Figure 7 is a screenshot of the user interface showing the ASV trajectory. The mission depicted in Fig. 7 has lasted 1499 seconds, and the ASV has intercepted three targets, and performed 5 patrols.

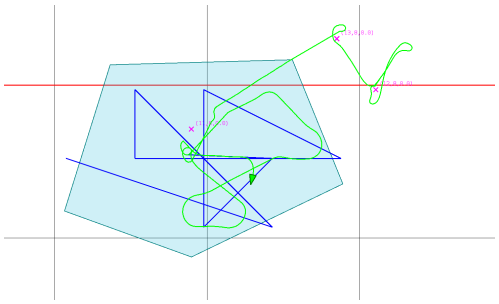


Fig. 7: Mission performed by the ASV. The green curve is the ASV trajectory, the green triangle the ASV position; pink crosses are intercepted targets; the zone in light blue is the area to protect, and the blue curve the computed patrol trajectory. Black squares are 1km wide. The red line is $y = 0$.

We have computed the state space of the Petri nets obtained from (10) and (11), using the simulation handler of Fig. 3 in place of the ROS handler in order to simulate possible success or failure of each skill. Then we have checked their soundness as discussed in Sect. III-F.2:

- they terminate: an exit marking (*i.e.*, one with the exit place marked) is always eventually reachable from the initial marking;
- they are clean: in every reachable exit marking, only the exit place is marked among the control flow places as specified by Eq. (9).

V. Conclusion

We have presented ASPiC, an acting system based on the representation of the elementary skills of a robotic system using Skill Petri nets, a Petri net model using control-flow semantics. ASPiC comes with composition operators that allow to build up action plans by using common operational constructs (sequence, concurrency, branches), managing failure cases using exception tokens. We have also presented and discussed some good properties to demonstrate on the resulting Petri net. ASPiC has been used to control an ASV performing a marine protection mission for more than 90 minutes, involving several actions to execute and several failures to manage. On this mission, we have used state space exploration to enforce the properties discussed before.

In future works, we aim at proving that the soundness property is guaranteed by construction, and we then have to correct the $N_{\otimes}$ operation net and base our proofs on the work in [18], [19].

We would also like to define a Domain Specific Language to help users define their own compositions without having to directly manipulate Petri nets, and eventually use this language to automatically translate planner's actions from a planning language (like PDDL) into ASPiC executable Petri nets.

References

[1] F. Ingrand and M. Ghallab, "Deliberation for autonomous robots: A survey," *Artificial Intelligence*, vol. 247, pp. 10–44, 2017.

[2] M. Ghallab, D. Nau, and P. Traverso, "The actor's view of automated planning and acting: A position paper," *Artificial Intelligence*, vol. 208, pp. 1–17, 2014.

[3] V. Verma, A. Jonsson, R. Simmons, T. Estlin, and R. Levinson, "Survey of Command Execution Systems for NASA Spacecraft and Robots," in *ICAPS Workshop on Plan Execution*, Monterey, CA, USA, 2005.

[4] V. Verma, T. Estlin, A. Jonsson, C. Pasareanu, R. Simmons, and K. Tso, "Plan Execution Interchange Language (PLEXIL) for Executable Plans and Command Sequences," in *i-SAIRAS*, Munich, Germany, 2005.

[5] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen, "A deliberative architecture for AUV control," in *ICRA*, Pasadena, CA, USA, 2008.

[6] J. Bohren and S. Cousins, "The SMACH High-Level Executive," *IEEE Robotics & Automation Magazine*, vol. 17, pp. 18–20, 2010.

[7] P. Schillinger, S. Kohlbrecher, and O. von Stryk, "Human-Robot Collaborative High-Level Control with an Application to Rescue Robotics," in *ICRA*, Stockholm, Sweden, 2016.

[8] H. Costelha and P. Lima, "Modelling, analysis and execution of robotic tasks using Petri nets," in *IROS*, San Diego, CA, USA, 2007.

[9] V. Ziparo, L. Iocchi, P. Lima, D. Nardi, and P. Palamara, "Petri Net Plans: A framework for collaboration and coordination in multi-robot systems," in *AAMAS*, Estoril, Portugal, 2008.

[10] W. van der Aalst, "The application of Petri nets to workflow management," *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[11] E. Best, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz, "M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages," *Acta Informatica*, vol. 35, no. 10, pp. 813–857, 1998.

[12] F. Pommereau, *Algebras of coloured Petri nets*. Lambert Academic Publishing, 2010.

[13] ——, "ABCD: A user-friendly language for formal modelling and analysis," in *Petri Nets*, Torun, Poland, 2016.

[14] S. Christensen and N. D. Hansen, "Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs," in *Petri Nets*, Chicago, IL, USA, 1993.

[15] R. Arkin, "Integrating behavioral, perceptual, and world knowledge in reactive navigation," *Robotics and Autonomous Systems*, vol. 6, no. 1–2, pp. 105–122, 1990.

[16] R. Firby and M. Slack, "Task Execution : Interfacing Networks to Reactive Skill Networks," in *AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, Palo Alto, CA, USA, 1995.

[17] R. Brafman, M. Bar-Sinai, and M. Ashkenazi, "Performance level profiles: A formal language for describing the expected performance of functional modules," in *IROS*, Daejeon, South Korea, 2016.

[18] E. Best, R. Devillers, and M. Koutny, *Petri Net Algebra*. Springer, 2001.

[19] R. Devillers, H. Klaudel, M. Koutny, and F. Pommereau, "Asynchonous Box Calculus," *Fundamenta Informaticae*, vol. 54, no. 1, 2003.

[20] D. Doose, C. Grand, and C. Lesire, "MAUVE Runtime: A Component-Based Middleware to Reconfigure Software Architectures in Real-Time," *JOSER*, vol. 8, no. 1, pp. 128–140, 2017.

[21] F. Pommereau, "SNAKES: A Flexible High-Level Petri Nets Library," in *Petri Nets*, Brussels, Belgium, 2015.