Zone-based Formal Specification and Timing Analysis of Real-time Self-adaptive Systems

Matteo Camilli[a,*], Angelo Gargantini[b], Patrizia Scandurra[b]

[a] *Dept. of Computer Science, Università degli Studi di Milano, Italy*
[b] *Dept. of Management, Information and Production Engineering (DIGIP), Università degli Studi di Bergamo, Italy*

Abstract

Self-adaptive software systems are able to autonomously adapt their behavior at run-time to react to internal dynamics and to uncertain and changing environment conditions. Formal specification and verification of self-adaptive systems are tasks generally very difficult to carry out, especially when involving time constraints. In this case, in fact, the system correctness depends also on the time associated with events.

This article introduces the *Zone-based Time Basic Petri nets* specification formalism. The formalism adopts *timed adaptation models* to specify self-adaptive behavior with temporal constraints, and relies on a *zone-based modeling* approach to support separation of concerns. Zones identified during the modeling phase can be then used as modules either in isolation, to verify *intra-zone properties*, or all together, to verify *inter-zone properties* over the entire system. In addition, the framework allows the verification of (timed) *robustness properties* to guarantee self-healing capabilities when higher levels of reliability and availability are required to the system, especially when dealing with time-critical systems. This article presents also the ZAFETY tool, a Java software implementation of the proposed framework, and the validation and experimental results obtained in modeling and verifying two time-critical self-adaptive systems: the Gas Burner system and the Unmanned Aerial Vehicle system.

Keywords: Self-adaptation, Real-time Systems, Petri Nets, Formal Verification, Timing Analysis

1. Introduction

Modern advanced software systems are required to perceive important structural and dynamic changes of their operational environment as well as of their internal status, and to adapt to these continuous changes autonomously [1, 2, 3]. They aim at achieving better quality of service and ensuring the required functionality in a *fail-soft* manner even in hostile or error conditions realizing the so called self-* properties [4], such as *self-optimizing* (when operating conditions change), *self-reconfiguration* (when a goal changes), *self-healing* (that allows the system to perceive that it is not operating correctly and therefore make the necessary adjustments to restore itself to normal operation autonomously), and so forth. Moreover, another main concern in the construction of self-adaptive software systems is the *uncertainty* [5, 6, 7] underlying the knowledge used for decision making. In fact, due to unpredictable changes, for example in the environment, a self-adaptive system may have no control over new unexpected processes that influence the environment and the system's organization itself that (especially in a distributed setting) may fluctuate dynamically.

The development of *self-adaptive systems* is extremely challenging and demands new formal approaches that can efficiently tackle the problems of expressing autonomicity requirements and ensuring the functional correctness of the system's adaptation logic both at design-time and at run-time. However, the survey

*Corresponding author.
Email addresses: matteo.camilli@unimi.it (Matteo Camilli), angelo.gargantini@unibg.it (Angelo Gargantini), patrizia.scandurra@unibg.it (Patrizia Scandurra)

in [8] and the Dagstuhl seminar [9] show that, although the attention for self-adaptive software systems is gradually increasing, the number of studies that employ mathematically-based specification and verification techniques from the area of formal methods remains low. In particular, specification and verification of self-adaptive systems are very difficult to carry out when involving time constraints. The functional correctness of the system and of its adaptation logic depends, in fact, also on the time associated with events. Most of the existing techniques are not effective when dealing with real-time constraints, because quantitative temporal aspects are not taken into account. Responsiveness is in fact a crucial property in real-time software systems, hence the need for models of timed adaptation that treat timing requirements (such as duration and overhead) of the adaptation as first class entities [1].

In this article, we present a formal framework to specify and verify the behavior of real-time self-adaptive systems. In particular, our main target systems are self-adaptive systems that exhibit a self-healing behavior [10] under time constraints. We define a specification formalism based on the *Time-Basic Petri nets* (or simply TB nets) [11], a particular timed extension of Petri nets. The proposed formalism, called *Zone-based TB Petri nets*, provides some enhancements to the TB nets formalism to model self-adaptive systems with real-time constraints. In particular, we adopt and extend the *adaptation models – one-point adaptation*, *overlap adaptation*, and *guided adaptation* – originally presented in [12, 13] to realize self-adaptation (SA) with temporal constraints in TB nets. To this purpose, the proposed specification formalism supports separation of concerns by allowing dividing the system's TB net model into *zones*[1]. The proposed framework supports a formal verification technique through the symbolic execution of the system's TB net, thus allowing also the verification of timed adaptation[2]. Zones of the TB net identified during the modeling phase can be used as modules (TB subnets) either in isolation, to check *intra-zone properties*, or all together, to check *inter-zone properties* of the overall system model.

The framework has been implemented as a JAVA software tool, called ZAFETY[3], and validated for modeling and verifying the self-healing behavior of time-critical systems. As running example to illustrate the framework, we present the Unmanned Aerial Vehicle (UAV) case study.

The definition of Zone-based TB nets and the supported verification technique to assure the correctness of these models were already presented in [15]. Starting from preliminary results presented in that work, the current contribution: (i) improves the definition of the framework to specify and verify the functional behavior of a self-adaptive system with real-time constraints; (ii) presents a more complex case study, a UAV system, showing all forms of adaptation models supported by our framework; (iii) describes the complexity evaluation of the main algorithms adopted by the proposed verification technique to assure correctness of a real-time self-adaptive system, including timed adaptation requirements; (iv) describes the current implementation of the proposed formal framework – the ZAFETY software tool – including some experimental results; (v) provides a more accurate comparison with the state of the art according to generally agreed key features for SA.

The proposed framework is primarily tailored to the formalization and timing verification of self-adaptive systems with time constraints in order to provide guarantees of correctness of adaptation at design time. Currently, we support the modeling of external uncertainty [5], mainly due to lack of knowledge on the environment. We do not consider modeling of non-functional requirements of self-adaptive systems, and we do not yet support runtime verification. Specifically, we contribute with the following key aspects: (i) definition of a formalism for modeling self-adaptive behavior with real-time constraints; (ii) modular and incremental modeling and verification process due to separation in zones (e.g., adaptation/functional logics); (iii) verification technique for checking adaptation requirements satisfaction, including timed adaptation; (iv) approach able to map zones defined upon the model into regions of the system state space, thus allowing the identification of different behaviors (normal behavior, undesired behavior, adaptive behavior, etc.) upon the state space structure, and the verification of properties of interest with respect to SA that, as

[1]The term zone is here to be intended differently from the forward zone-based reachability analysis of Time Petri Nets [14], where a zone stands for a finite convex union of regions (i.e., a representation of clock values using equivalence classes).

[2]Timed adaptation concerns how specifying time-triggered adaptation coupling the design of adaptation concerns with time-dependent constraints. Support for modeling time must regard sparse and dense models for time separately.

[3]Zone-based specificAtion and veriFication of rEaltime self-adapTive sYstems.

advocated in [8], typically map to transitions between different zones. These aspects, as discussed in Sect. 7, are advantages over previous works that do not cover all these features or cover them only partially.

This article is organized as follows. Sect. 2 provides background concepts on the TB Petri nets and also describes the adaptation models that we adopt for realizing SA in TB Petri nets. Sect. 3 presents the UAV system adopted throughout the article as running example. Sect. 4 presents the proposed zone-based TB nets formalism for modeling real-time self-adaptive systems. Sect. 5 presents a verification technique to verify structural and behavioral properties of real-time self-adaptive systems, and the results of applying this verification technique on the running case study. Sect. 6 presents the tool ZAFETY and the experimental results obtained during the verification of two case studies, including the UAV system. Sect. 7 presents related works with respect to formal methods for modeling and analyzing time-dependent self-adaptive systems. Sect. 8 discusses some threats to the validity of our work. Sect. 9 concludes the article and outlines future directions of our work. Finally, Appendix A reports the complete formal specification of the UAV example.

2. Background

This section provides background concepts on the TB net formalism and existing adaptation models that we adopted as reference models to realize SA in TB nets.

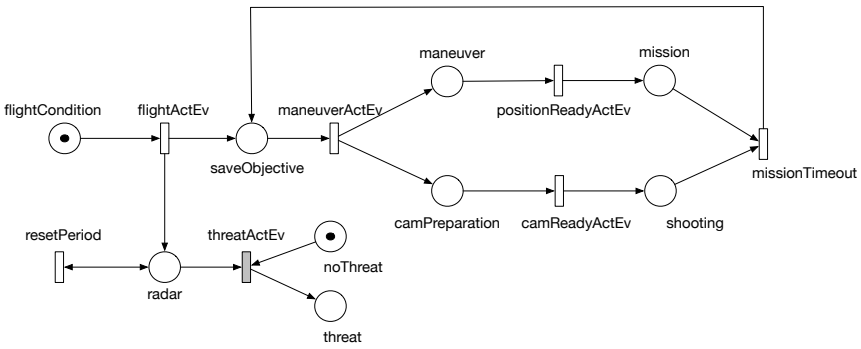
2.1. Time Basic Petri nets

TB nets belong to the category of Petri nets [16] in which system time constraints are expressed as linear functions associated to each transition, representing possible firing instants computed since transition's enabling. Tokens, atomically produced by the firing of a transition, are thereby associated to time-stamps with values ranging over a determined set. With respect to the well-known representative of this category, (i.e., Time Petri nets [17]), interval bounds in TB nets are linear functions of timestamps in the enabling marking, rather than simple numerical constants. We chose to adopt such a modeling formalism because it supports both time and functional extensions in a semantically clear and rigorous way. Thus it represents an effective formal model to deal with specification of time-critical systems. Although other modeling formalisms such as timed-automata [18] or finite-state-machines [19] support the modeling of temporal or behavioral aspects, PNs-based approaches can be more concise [20] and more scalable with respect to other formalisms for specifying highly concurrent systems [21]. Furthermore, aspects such as messaging, communication protocols, which are commonly used in concurrent or distributed systems, can be difficult to model with the language primitives of timed-automata [22, 21]. Moreover, TB nets are supported by powerful off-the-shelf open source software tools able to perform simulation, model checking [23, 24, 25, 26], model-based testing, and runtime verification [27, 28].

The structure of a Time Basic net extends the P/T net structure (P, T, F), where P is a finite set of places, T a finite set of transitions such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ a set of arcs (or flows) connecting places to transitions and transitions to places. Let $v \in P \cup T$: ${}^\bullet v$, $v^\bullet$ denote the backward and forward adjacent sets of v according to F, respectively, also called pre/post-sets of v. A time-stamp *tuple* of $t \in T$ is an association $en : {}^\bullet t \to \mathbb{R}_{\geq 0}$. Moreover, each transition t is associated with a *time function* f_t which maps a tuple of time-stamps en of t to a (possibly empty) set of $\mathbb{R}_{\geq 0}$ values. $f_t(en)$ represents the possible firing times of t.

A *marking* (state) is a mapping $m : P \to Bag(\mathbb{R}_{\geq 0})$, where $Bag(X)$ represents the set of multiset over X. According to the *weak* (or *non-urgent*) semantics [11, 29, 30], t can fire at any instant $\tau \in f_t(en)$. The *strong* (or *urgent*) interpretation [11, 31] states that t must fire at an instant $\tau \in f_t(en)$, unless it is disabled by the firing of any conflicting transitions at an instant no greater than the latest firing time of t. Letting en be an enabling tuple of t, a pair (en, τ), $\tau \in f_t(en)$, is said a *firing instance* of t. The firing of (en, τ) produces the new marking m', such that:

- $\forall p \in {}^\bullet t \setminus t^\bullet \; m'(p) = m(p) - en(p)$
- $\forall p \in t^\bullet \setminus {}^\bullet t \; m'(p) = m(p) + \{\tau\}$



Initial marking: `flightCondition`$\{T_0\}$ `noThreat`$\{T_0\}$
Initial constraint: $0 \leq T_0 \leq 10$

Time-Functions:

`flightActEv`	[`flightCondition` + 0.5, `flightCondition` + 2.0]
`threatActEv`	[`radar` + 30.0, `radar` + 30.5]
`resetPeriod`	[`radar` + 30.0, `radar` + 30.5]
`maneuverActEv`	[`saveObjective` + 1.0, `saveObjective` + 60.0]
`positionReadyActEv`	[`maneuver` + 0.5, `maneuver` + 60.0]
`camReadyActEv`	[`camPreparation` + 0.5, `camPreparation` + 1.5]
`missionTimeout`	[`mission` + 0.2, max(`mission` + 0.2, `shooting`) + 60.0]

Figure 1: Simplified UAV TB net model. Weak transitions are depicted in gray.

- $\forall p \in t^\bullet \cap {}^\bullet t \; m'(p) = m(p) - en(p) + \{\tau\}$
- for all remaining places, $m'(p) = m(p)$

Hereafter, we denote a time function f_t with a pair of linear functions $[lb_t, ub_t]$, denoting parametric interval bounds.

An example of TB net to illustrate concepts is shown in Figure 1. It models a simplified fragment of the UAV system's behavior described later in Sect. 3. The places in this TB net denote the main phases and high level actions in progress of the system. The initial marking represents the system flying after the takeoff phase (the marked place `flightCondition`). Once the `flightActEv` transition fires, the UAV starts an observation mission in a dynamic, partially known and unsafe environment. The observation mission starts by setting a new objective (the marked place `saveObjective`) and proceeds by performing the proper maneuver and reaching the point of interest (transition `maneuverActEv` and transition `positionReadyActEv`, respectively). At the same time, the UAV prepares the camera to perform the photo shoot operation (transition `camReadyActEv`). The time function associated with the transition `missionTimeOut` can be interpreted as follows. It cannot fire before 0.2 time units since the appearance of a token in the place `mission` (i.e., the minimum time required by a very short photo shoot operation). Moreover, the firing time cannot exceed 60 time units plus the maximum between the time-stamp of the token in the place `shooting` and the time-stamp of the token in the place `mission` plus 0.2.

During the observation mission, the flight path can intersect any unknown entity that represents a threat such as hostile presences but also bad weather areas or no-fly zones. Thus the system is equipped with a radar (the marked place `radar`) sensed every 30 time units. Once the radar is sensed, the system can either reset the sensing period, or recognize a threat. This latter case is modeled by the *weak* transition `threatActEv`. Whenever a threat is established, a recovery procedure should take control of the system in order to avoid invalid behavior in this degraded situation. However, this procedure is not modeled in this simplified fragment of the UAV system.

2.2. Time Reachability Graph

The Time Reachability Graph (*TRG*) [32] is an abstract state-transition system which covers the infinite state space associated with a TB net model. The *TRG* construction, implemented by the GRAPHGEN

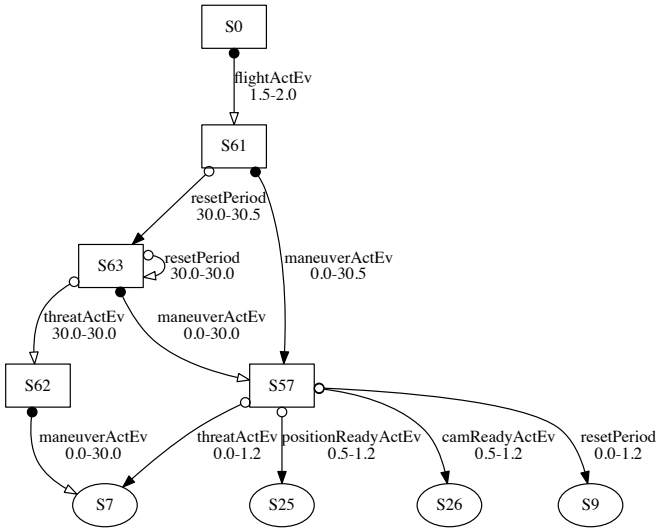


Figure 2: A portion of the time reachability graph constructed from the simple UAV model in Figure 1.

software tool [32, 33], relies on the *symbolic state* notion and the *symbolic firing* mechanism.

Let $TS = \{T_i\}_{i \geq 0}$, be the set of time-stamp symbols. A TRG node is a *symbolic state* $S = \langle M, C \rangle$, where M (symbolic marking) maps places into elements of $Bag(TS)$ and C (constraint) is a logical predicate formed by linear inequalities defined in terms of $TS \cup \{T_L\}$, where the special symbol T_L represents the state creation instant (i.e., the last time-stamp produced in S). The symbolic constraint C contains relative time dependencies between timestamps.

For instance, the initial symbolic state of the simple UAV (Figure 1) is represented by $S_0 = \langle M_0, C_0 \rangle$, where:

$$\begin{array}{rcl} M_0 & := & \texttt{flightCondition}\{T_0\}, \texttt{noThreat}\{T_0\} \\ C_0 & := & T_0 \geq 0 \ \wedge \ T_0 \leq 10 \end{array}$$

An ordinary marking m belongs to $S = \langle M, C \rangle$ if and only if m is obtained from M through a numerical replacement of time-stamps being a solution of C.

We say that $S \subseteq S'$ (where $S = \langle M, C \rangle$ and $S' = \langle M', C' \rangle$), if and only if all the ordinary markings belonging to S belong also to S'. A sufficient condition for $S \subseteq S'$ is $M = M'$ and $C \Rightarrow C'$ [4].

The *TRG* construction starts from the initial symbolic state S_0. Whenever a successor S' of S is generated, we check the existence of any node S'' such that either $S' \subseteq S''$ or $S' \supset S''$ (in the latter case S' absorbs S'').

During the *TRG* building we can recognize timestamp symbols that have been shown irrelevant for the model's time evolution from a given state. Thus, they are replaced in the marking by T_A symbols (i.e., *anonymous time-stamps*), then eliminated from the constraint [32, 29].

Hereafter, we refer to the *TRG* structure, computed from the model $PN = \langle P, T, F \rangle$, with the notation $TRG(PN) = \langle N, E, S_0 \rangle$, where N is the set of reachable symbolic states, $E \subseteq N \times N$ is the set of symbolic edges, and S_0 is the initial symbolic state.

The *TRG* edges are labeled differently, depending on the property which has been assessed between a state S and its successor S'. The possible relations between directly connected symbolic states are reported

[4]The Floyd-Warshall [34] and the Simplex [35] algorithms are used for variable elimination and satisfiability checking, respectively.

below. Let e be the edge relation of the underlying ordinary state space. Every symbolic edge satisfies the condition **EE** ($\bigcirc\!\!-\!\!\triangleright$ [5]) [36, 37], i.e.:

- $\forall\langle S, S'\rangle \in E,\ \exists m \in S, \exists m' \in S' : \langle m, m'\rangle \in e$
- $\forall\langle m, m'\rangle \in e,\ \forall S \in N : m \in S,\ \exists S' \in N : m' \in S' \wedge \langle S, S'\rangle \in E$

The first part avoids two abstract states from being connected, if no corresponding ordinary states are. The second part ensures that each path in the ordinary state space has a representative in the abstract one. GRAPHGEN is able to (locally) recognize stricter conditions: in such cases *TRG* arcs are suitably labelled, as follows. Let $\langle S, S'\rangle \in E$.

$$\begin{array}{lll} \mathtt{EA}\ (\bigcirc\!\!-\!\!\blacktriangleright) & \iff & \forall m' \in S', \exists m \in S : m \to m' \\ \mathtt{AE}\ (\bullet\!\!-\!\!\triangleright) & \iff & \forall m \in S, \exists m' \in S' : m \to m' \\ \mathtt{AA}\ (\bullet\!\!-\!\!\blacktriangleright) & \iff & \mathtt{EA} \wedge \mathtt{AE}, \end{array}$$

where the symbol $\to$ represents the one-step reachability relation between ordinary markings, i.e., m' is reachable from m by firing a single transition.

Figure 2 shows a portion of the *TRG* computed from the model in Figure 1, using GRAPHGEN. For the sake of readability we do not show the entire structure (composed of 41 reachable symbolic states and 81 edges). Elliptic states have different outgoing edges (not shown in Figure 2) connecting them to other elements of the graph. The edges between symbolic states are labeled with the firing transition and two numeric values representing the local minimum and the local maximum temporal distance between directly connected symbolic states, respectively. Edges can be of different types following the above notation. For example, the **AE** edge $\langle S_0, S_{61}\rangle$ declares that only a subset of the concrete states represented by S_{61} is reachable from S_0.

2.3. Adaptation models

This Section discusses the main characteristics of the state space (i.e., the *TRG* in case of TB nets) of a system exhibiting self-adaptive behavior.

The *TRG* reported in Figure 2 shows the behavior over time of the simple UAV example operating in its own main domain (i.e., the observation mission). In general, a self-adaptive system operates in different domains and changes its behavior at run-time in response to changes of the domain. Therefore, its reachable states can be separated into disjoint regions each of which operates in a different domain and exhibits a different *steady-state* behavior [38, 13].

Figure 3 depicts the simplified state space of a self-adaptive system. $\mathcal{S}$ and $\mathcal{T}$ are two regions representing the system operating in two different domains, while $\mathcal{A}$ (i.e., the *adaptation set*) represents the set of states and edges connecting $\mathcal{S}$ to $\mathcal{T}$. Since we address real-time self-adaptive systems, we enrich the adaptation set with a temporal constraint τ, in order to ensure the adaptation within a proper temporal deadline. The adaptation set between the source and the target domains can describe different kind of adaptive behaviors.

Three common types of adaptive behavior are: *one-point adaptation*, *guided adaptation*, and *overlap adaptation* [12, 13].

One-point adaptation. The *one-point adaptation* process completes with a single edge e (i.e., the adaptation set $\mathcal{A}$ contains only the e element). Thus the steady-state behavior $\mathcal{S}$ ends within e and the steady-state behavior $\mathcal{T}$ starts immediately after e [12, 13]. The source states suitable for adaptation, with outgoing adaptive transitions, are called *quiescent states*.

[5] A visual representation of a TRG arc satisfying the **EE** property.

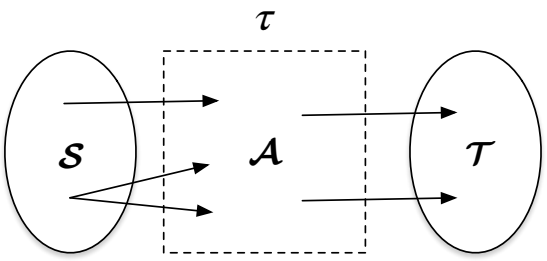


Figure 3: Two regions and an adaptation set with time constraints upon transitions.

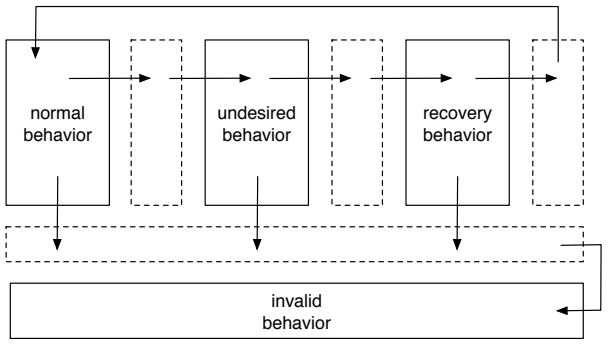


Figure 4: State space regions denoting different behaviors of a self-adaptive system.

Overlap adaptation. During the *overlap adaptation* process, the target behavior starts immediately after reaching $\mathcal{A}$, and the source behavior stops when the system leaves $\mathcal{A}$ and enters the region $\mathcal{T}$. At this point the system exhibits only the target steady-state behavior [12, 13]. This means that within the boundaries of $\mathcal{A}$ the source and the target behaviors overlap. The adaptive system must always satisfy the *adaptation integrity constraint*: once the adaptation process starts, it should eventually end and reach a state belonging to the target region. Violations of this constraint result in an inconsistent state of the system that is not designed for the target region, and there are no means to ensure its correctness.

Guided adaptation. The *guided adaptation* process starts when the system receives an adaptation request in a non quiescent state. Therefore, the system should enter a restricted mode, where some of its functionalities are usually blocked in order to reach a quiescent state and perform the adaptation as fast as possible [12, 13]. In this case, from the region $\mathcal{S}$ the system reaches $\mathcal{A}$ that represents the restricted mode. Entering the restricted mode $\mathcal{A}$ ensures that the system will reach a quiescent state, from which one-point adaptation takes the system to $\mathcal{T}$. To specify a guided adaptation, we should determine the functionalities that should be blocked in the restricted mode, and identify the quiescent states of the system in the restricted mode.

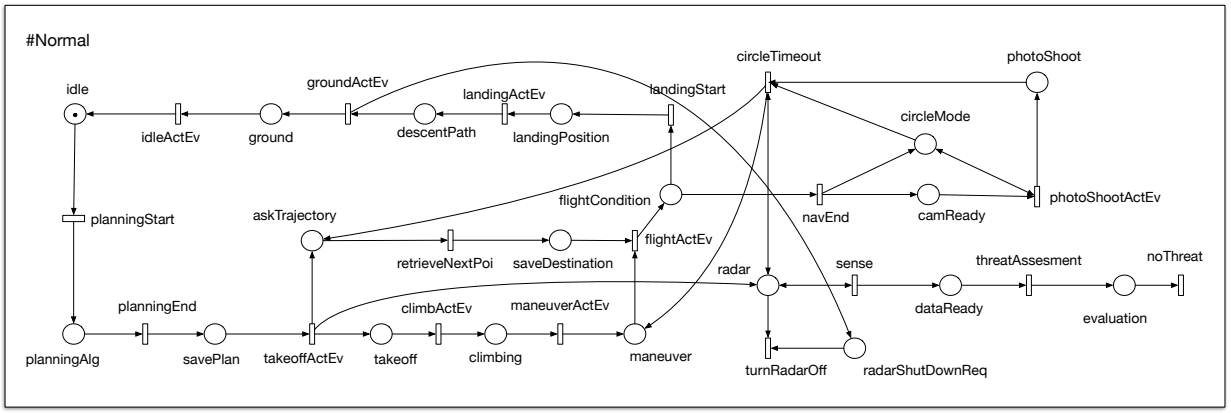
In section 4, we incorporate these adaptation models into the TB nets formalism and extend them to support timing constraints and different temporal semantics.

3. A running example: the Unmanned Aerial Vehicle

UAVs [39] usually fly in a very dynamic environment that requires dynamic changes to the flight plan, therefore it represents a very common real-time self-adaptive system example. In fact, the usage of UAVs over conventional aircraft has many advantages [40], such as reduced operating costs and better flight performance. Moreover, there is a strong interest in UAVs for military applications, although civilian applications are expected to become more common [41], as the UAV technology advances.

Our interest is to model and verify the management of an observation mission for UAVs in a dynamic, partially known and unsafe environment. The objectives of the observation mission are to join several points of interest and to carry out a photo shoot operation there. The typical phases of this mission are:

- planning computation and takeoff,
- navigation to the next point of interest until they are all visited,
- operation on the mission area,
- return to the ground station and landing.



Initial marking: `idle`$\{T_0\}$
Initial constraint: $0 \leq T_0 \leq 10$

Time-Functions:

`planningStart`	[`idle`, `idle`+∞]	`planningEnd`	[`planningAlg` + 0.2, `planningAlg` + 0.5]
`takeoffActEv`	[`savePlan` + 1.5, `savePlan` + 2.0]	`climbActEv`	[`takeoff` + 0.5, `takeoff` + 0.8]
`maneuverActEv`	[`climbing` + 10, `climbing` + 100]	`retrieveNextPoi`	[`askTrajectory`, `askTrajectory` + 0.5]
`landingStart`	[`flightCondition`, `flightCondition`+∞]	`flightActEv`	[`maneuver`, `max`(`maneuver`, `saveDestination`) + 1.2]
`sense`	[`radar` + 30, `radar` + 30.5]	`turnRadarOff`	[`radar` + 0.1, `max`(`radar`, `radarShutdownReq`) + 0.5]
`threatAssesment`	[`dataReady` + 0.5, `dataReady` + 0.8]	`noThreat`	[`evaluation` + 1.5, `evaluation` + 1.8]
`photoShootActEv`	[`cameraReady` + 0.5, `cameraReady` + 0.8]	`circleTimeout`	[`photoShoot` + 60, `max`(`photoShoot` + 60.5, `radar`)]
`idleActEv`	[`ground`, `ground` + 1.5]	`landingActEv`	[`landingPosition` + 0.5, `landingPosition` + 0.8]
`groundActEv`	[`descentPath` + 10, `descentPath` + 100]	`navEnd`	[`flightCondition` + 10, `flightCondition`+∞]

Figure 5: The nominal behavior of the UAV case study.

During the nominal behavior, the flight path can intersect any unknown entity that represents a threat such as hostile presences but also bad weather areas or no-fly zones. Therefore the UAV should adapt itself and eventually change the flight plan in order to take a detour around the threats.

Figure 5 shows the nominal behavior of the UAV system, i.e., the general behavior of the vehicle from its takeoff until its landing. The places indicate the main phases of the mission and the high level actions in progress. The meaning of the box named `#Normal` will be discussed in section 4.

The initial marking represents the idle phase, where the brakes maintain the vehicle still on ground until the mission starts. During this phase, a real system should also perform some pre-flight tests. At the beginning of the mission, before the takeoff phase, the system starts the first calculation of the plan through the planning program. The purpose of the planning program is to build a feasible flying path by taking into account the constraints of the vehicle (e.g., fuel, flight maximum altitude, etc.), the environment (e.g., weather areas, no-fly zones, etc.) and the mission (e.g., mission range, position of points of interest, etc.). Once the trajectory has been saved, the takeoff phase starts (the transition `takeoffActEv` fires). Once the expected takeoff position is reached, the UAV is steered in the runway direction and then full throttle is set until the takeoff speed is reached. When the transition `climbActEv` fires, the takeoff is considered finished and the climbing phase starts. The climbing phase keeps the vehicle in the takeoff direction and sets a fixed climb rate that allows it to reach a desired altitude. During the climbing, the first point of interest (POI) is retrieved and then saved as destination. When the right altitude is reached (the transition `maneuverActEv` fires), the maneuver phase takes the UAV in the direction of the first POI. After the maneuver, a flight condition is entered (the transition `flightActEv`), and the travel phase begins. It basically sets a great-circle route (shortest distance between two points on a sphere) between the current position and the intended destination, at the specified speed and altitude. After the target position is reached (the transition `navEnd` fires), a circle mode waypoint is set: four waypoints forming a diamond are calculated around the target

Requirement	Description	Type
R1: Path optimization	The planning program must select a flying path minimizing the difference between the costs and the revenues.	Performance, energy
R2: Timed takeoff	Once the trajectory has been computed, it is always possible to reach the takeoff phase in t_1 time units.	Liveness, bounded-response time
R3: Timed threat sensing	Once the radar is sensed, either a threat is detected in t_2 time units, or no threat is detected.	Bounded-response time
R4: Emergency condition	When the UAV is in flight condition, a recovery procedure must always follow an undesired situation to avoid an emergency condition.	Safety
R5: Timed threat avoidance	Once a threat has been established, the recovery procedure must adjust the flight plan taking at most t_3 time units. The distance between the UAV and the threat area center must be sufficient to avoid intersection.	Safety, robustness, bounded-response time
R6: Timed landing	Once the last POI has been reached, the UAV must always start a landing phase in t_4 time units.	Liveness, bounded-response time

Table 1: UAV adaptation requirements.

position. The UAV then cycles through those in a clockwise (or anti-clockwise) direction until the specified time limit is reached. During the circle mode, the UAV prepares the camera and actuates a photo shoot to gather information about the current POI. When the time limit is reached (the transition `circleTimeout` fires), the next POI is retrieved and a new maneuver and then a new travel phase is executed. Whenever the UAV ends the main mission and reaches the expected landing position (the transition `landingStart` fires), the UAV descends at a specific angle, then performs a flare maneuver when close to the ground, and finally stops when ground contact has been established (the transition `groundActEv` fires). After the landing phase, the UAV returns in idle mode.

In order to make the UAV system robust to possible exceptions of its operating context, we want to endow it with a SA logic [42], i.e., we want to add a *self-healing* behavior [43, 44] in charge of handling disruption and restore the system to normal conditions within hard deadlines. Therefore, the system should be able to diagnose itself and react to faults or failures within strict time constraints such that a satisfactory mode of operation is restored avoiding great loss, such as damaging the surrounding physical environment or even threatening human lives. Table 1 lists some examples of adaptation requirements, typically found in this application domain. The next section introduces our zone-based TB net formalism and shows its use for modeling the adaptive behavior of the UAV during its mission.

4. Zone-based TB nets

The TB net shown in Figure 5 models the nominal behavior of the UAV without any exception (e.g., hostile presences or bad weather conditions). In order to endow the UAV system with the ability to (automatically) diagnose and fix in real-time various problems arising in its operating context, a self-healing subsystem must be added to the UAV to detect undesired behavior and then adapt itself to restore the normal behavior within strict time constraints.

Inspired by the model introduced in [8], the state space of a self-adaptive system should be characterized by different disjoint regions representing the steady-state behaviors of the system. These regions, sketched by Figure 4, are the *normal behavior*, where the system performs its main functionality without failures; the *undesired behavior*, which represents an exception where adaptation is required in order to avoid invalid states to be reached; the *recovery behavior*, where the system adapts itself to deal with the undesired behavior; and the *invalid behavior* that represents all the states where the system should never be (e.g., deadlocks or loss of functionality). Among these regions, "adaptation sets" (sets of states and arcs represented by dashed

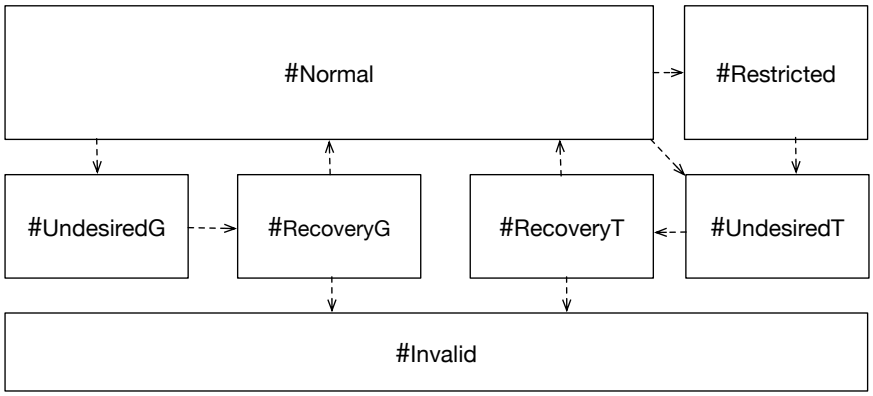


Figure 6: High-level schema of the zone-based model of the UAV case study.

lines in Figure 4 and connecting regions) captures the adaptation logic of the system and therefore carry out the behavioral adaptation of the system controller from the source to the target domains using different adaptation models (as introduced informally in Sect. 2.3).

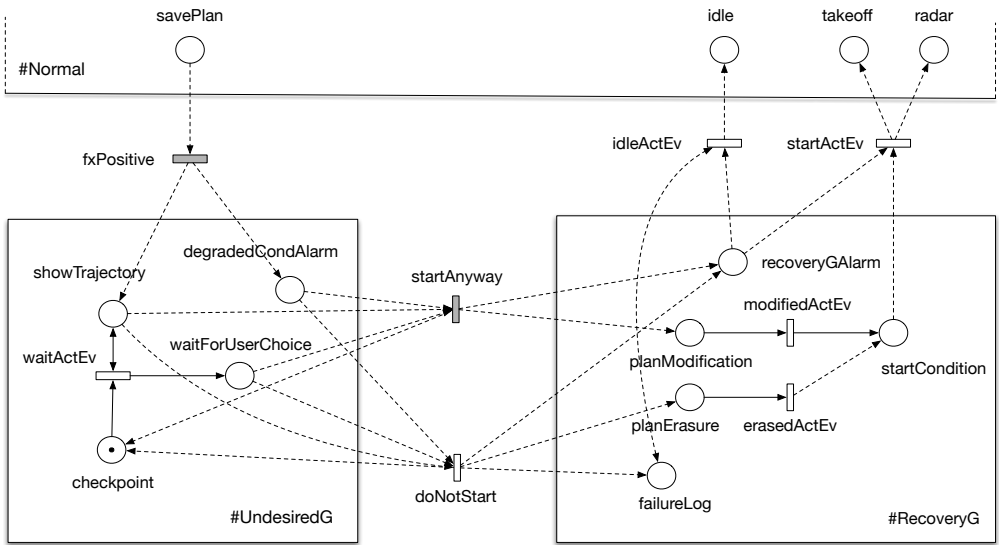
In order to be able to build the state space of a self-healing system with these structural properties, we propose a *zone-based modeling* approach using the TB net formalism. This technique aims at identifying and isolating different modules (i.e., TB sub-nets) of the system representing different steady-state behaviors. Source modules and target modules should not include information about each other, or about the adaptation. The source and target modules should be verified against the local requirements for the source and target domains, respectively, and then all together along with the adaptation models to verify adaptation requirements. To this end, zones are used to identify different state space regions and adaptation sets, used in turn to verify the proper correctness properties. The following subsections exemplify and formalize this approach. First, Sect. 4.1 directly introduces the formalism trough the UAV case study, and then Sections 4.2, 4.3 and 4.4 formalize the underlying concepts.

4.1. Zone-based specification of the UAV system with self-healing capabilities

Figure 6 shows the high-level schema of the zones of the UAV model with self-healing capabilities. The zones are represented by white boxes named using the notation `#<ZoneName>`. Zones are interconnected by different *adaptation models* represented by dashed arrows. The zone-based specification allow us to deal with different mission concerns separately and to integrate new adaptive behaviors easily and in a modular way. Specifically, the UAV system can adapt itself in two different phases of the zone `#Normal` to change the flight plan and avoid invalid behavior. Path adjustments during the mission evaluation (while the UAV is still on ground) are modeled by the zones `#UndesiredG` and `#RecoveryG`, while the path adjustments during the travel are modeled by the zones `#Restricted`, `#UndesiredT`, and `#RecoveryT`. The zone `#Invalid` models loss of functionality from which the recovery is no longer possible.

The complete zone-based TB net of the UAV system is reported in Appendix A. It details the vehicle behavior during the mission in nominal mode (normal zone) and in the degraded situations (zones undesired, restricted, recovery, and invalid). Below, we describe separately the different TB net zones of the UAV model. Intuitively, zones (e.g., see the zones `#UndesiredG` and `#RecoveryG` shown in Figure 7) are compositions of places and transitions grouped within white boxes. Cross-zone transitions, i.e. the transitions with incoming/outgoing dashed arcs outside the boundaries of the boxes, realize the mechanism of "adaptation sets". We will formalize the concepts of zones and cross-zone transitions later in this Section.

- *The zone* `#Normal`: It corresponds to the TB net shown in Figure 5. It represents the general behavior of the vehicle from its takeoff until its landing as described in Section 3. Beside the nominal behavior, the UAV can exhibit different undesired ways of acting. For instance, before the takeoff and during the first calculation of the plan (the marked place `planningAlg`), the planning program has to select and order the



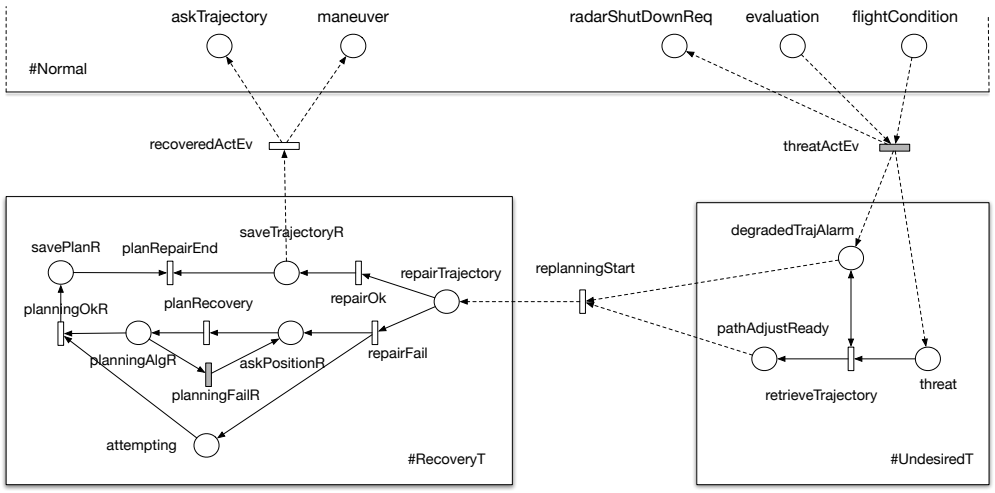
Time-Functions:

`fxPositive`	[`avePlan` + 0.2, `savePlan` + 0.5]	`waitActEv`	[`askPosition` + 0.1, `askPosition` + 0.5]
`startAnyway`	[`planningAlg`, `planningAlg` + 1.2]	`doNotStart`	[`showTrajectory`, `showTrajectory` + 0.1]
`modifiedActEv`	[`planModification`, `planModification` + 0.5]	`erasedActEv`	[`planErasure`, `planErasure` + 0.5]
`idleActEv`	[`failureLog` + 0.5, `failureLog` + 1.0]	`startActEv`	[`startCondition` + 0.1, `startCondition` + 0.5]

Figure 7: The `#UndesiredG` and `#RecoveryG` zones of the UAV system, along with cross-zone transitions. Weak transitions are depicted in gray.

best sub-set of objectives and to determine the arrival date at each POI, minimizing the difference between the costs and the revenues obtained on the selected path (i.e., a costs-revenues criterion). The costs include the consumption, the danger, and the duration of the mission. The total revenue is a reward associated with information gathered from the objective areas. If the costs are lower than revenues, the path is accepted and saved to start the mission. On the contrary, if the criterion is positive, the system has to handle this exceptional behavior, represented by the zone `#UndesiredG`.
After the takeoff, during the execution of the main mission (the marked places `circleMode` and `photoShoot`), the radar of the UAV continuously verifies whether the current flight path intersects any unknown threat. The radar is sensed every 30 time units and after the data became ready (the marked place `dataReady`) the threat evaluation starts. The evaluation algorithm calculates the shortest distance between the threat area center and the flight path, and then compares it to the threat area radius to determine whether they intersect or not. If an intersection is detected, the UAV enters a degraded situation (the zone `#UndesiredT`) that should be handled by carrying out the proper operations in charge of changing the flight plan in order to take a detour around the threat. If the threat is detected in a non quiescent state, the UAV should interrupt the operation in progress (through the zone `#Restricted`) as fast as possible and bring itself into a state from which the adaptation can be safely applied.

- *The zone* `#UndesiredG`: It represents a degraded situation that can manifest before the takeoff. If the costs-revenues criterion obtained on the selected path is positive (transition `fxPositive`), the system moves into the zone `#UndesiredG` represented by the corresponding TB net shown on the left of Figure 7. This undesired behavior simply models the user interaction that has to choose either to start anyway the mission (transition `startAnyway`) or abandon the mission (transition `doNotStart`).
- *The zone* `#RecoveryG`: Once the zone `#RecoveryG` (see Figure 7) is accessed, the recovery procedure allows the plan to be modified or erased, depending on the user choice. In the latter case the recovery procedure logs the failure. If the maximum number of retries (i.e., planning program failures) is reached, the system accesses the zone `#Invalid`. Otherwise, the system goes back into the nominal behavior by firing



Time-Functions:

`planningFailR`	$[\texttt{planningAlgR} + 1.5, \texttt{planningAlgR} + 1.5]$	`repairFail`	$[\texttt{repairTrajectory} + 1.5, \texttt{repairTrajectory} + 1.5]$
`planRepairEnd`	$[\texttt{savePlanR} + 0.5, \texttt{savePlanR} + 0.8]$	`repairOk`	$[\texttt{repairTrajectory} + 1.2, \texttt{repairTrajectory} + 1.5]$
`planningOkR`	$[\texttt{planningAlgR} + 1.2, \texttt{planningAlgR} + 1.5]$	`planRecovery`	[askPositionR+0.02, askPositionR+0.05]
`retrieveTrajectory`	$[\texttt{threat} + 0.8, \texttt{threat} + 1.2]$	`recoveredActEv`	$[\texttt{trajectorySavedR} + 0.1, \texttt{trajectorySavedR} + 0.5]$
`threatActEv`	$[\max(\texttt{evaluation}, \texttt{flightCondition}) + 1, \max(\texttt{evaluation}, \texttt{flightCondition}) + 1.8]$		

Figure 8: The `#RecoveryT` and `#UndesiredT` zones of the UAV system, along with cross-zone transitions. Weak transitions are depicted in gray.

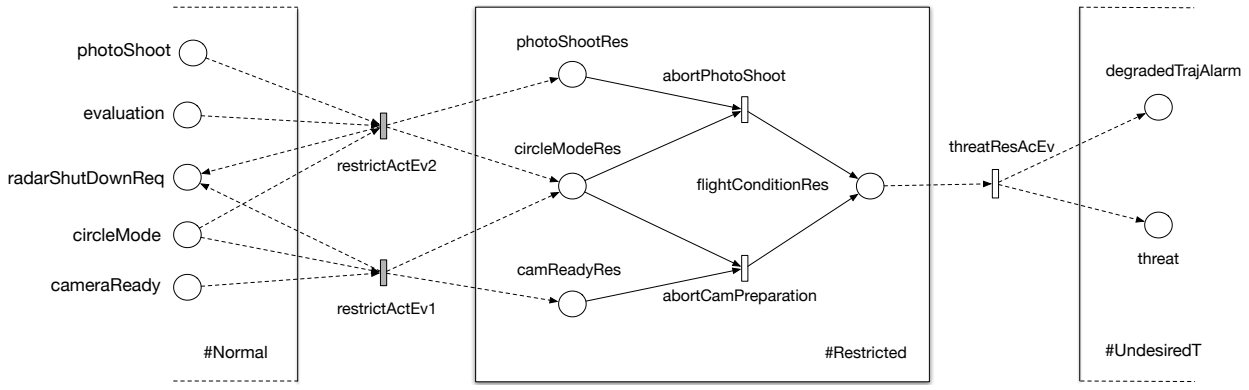
either the transition `startActEv` or the transition `idleActEv`.

• *The zone* `#UndesiredT`: It represents a degraded situation that can manifest after the takeoff. Once the threat has been established, the zone `#UndesiredT` (see Figure 8) raises an alarm and it retrieves the current trajectory to be passed to the recovery procedure. The transition `replanningStart` makes the system able to access the zone `#RecoveryT`, where the system tries to recover by adding a new waypoint to the flight plan.

• *The zone* `#RecoveryT`: Once the UAV adapts into this module (see Figure 8), it starts to adjust the flight path to avoid the current threat. In particular, a new waypoint is placed along the perpendicular to the original flight path passing through the threat area center, at a distance sufficient to avoid intersection. If the recovery procedure succeeds (transition `planningOkR` fires), the new flight plan is saved and the system goes back into the zone `#Normal` (transition `recoveredActEv` fires). If the recovery procedure fails (i.e., the attempting time limit ends), the system turns itself into the zone `#Invalid` trough the transition `attemptTimeout`, where the system cannot be recovered anymore (see Figure 10).

• *The zone* `#Restricted`: Whenever an occurring threat is detected in a non quiescent state of the nominal behavior, the UAV should interrupt the operation in progress as fast as possible and bring itself into a state from which the adaptation can be safely applied. The zone `#Restricted` (see Figure 9) models the interruption of different operations currently in execution (e.g., camera preparation through the firing of the transition `abortCamPreparation`, or photo shoot operations through the firing of the transition `abortPhotoShoot`) in order to bring the system into a state suitable for adaptation (i.e., the marked place `flightConditionRes`).

• *The zone* `#Invalid`: It represents loss of functionality (see Figure 10) from which the system recovery is no longer possible. The zone is reachable from degraded situations when the UAV is either on ground or after the takeoff (i.e., from the zones `#UndesiredG` and `#UndesiredT`, respectively). In particular, the invalid behavior is accessed from the zone `#RecoveryG` (through the firing of the transition `tooManyFailures`) whenever the maximum number of planning program failures has been reached. Therefore, the UAV shuts down itself (the transition `shutdownActReq`) and it requires human interaction to be restarted. Otherwise,



Time-Functions:
`abortCamPreparation` [`camReadyRes`, `camReadyRes` + 0.2] `abortPhotoShoot` [`photoShootRes`, `photoShootRes` + 0.2]
`restrictActEv` [max(`evaluation`, `operationReady`) + 1.0, max(`evaluation`, `operationReady`) + 1.8]
`restrictActEv2` [max(`evaluation`, `cameraReady`) + 1.0, max(`evaluation`, `cameraReady`) + 1.8]

Figure 9: The `#Restricted` zone of the UAV system, along with cross-zone transitions. Weak transitions are depicted in gray.

if the invalid behavior is accessed from the zone `#RecoveryT`, an occurring threat cannot be avoided (the marked place `emergencyCondition`). Thus the UAV detonates itself (the transition `detonationActEv` fires) before the impact to avoid possible damages to the surrounding environment or possible threats to human lives.

4.2. Zone-based TB nets formalization

Intuitively, a *zone* is a subnet of the entire model such that all its elements are connected only to elements of the same zone, except for transitions belonging to its own preset or postset allowing the connection among different zones. The following definitions formalize this intuition.

Definition 1. ***Subnet (and subnet preset/postset).*** *Given a TB net $\langle P, T, F\rangle$, a subnet is a triplet $\langle P_S, T_S, F_S\rangle$ s.t. $P_S \subseteq P$, $T_S \subseteq T$, $F_S \subseteq F$, $P_S \neq \emptyset$, $T_S \neq \emptyset$, and $F_S \neq \emptyset$. Given a subnet z, the preset ${}^\bullet z$ is the set of transitions that connect places outside z to places belonging to z. Given a subnet z, the postset $z^\bullet$ is the set of transitions that connect places of z to places outside z.*

Definition 2. ***Zone.*** *Given a TB net $\langle P, T, F\rangle$, a zone is a subnet $z = \langle P_S, T_S, F_S\rangle$ s.t. the structure $\langle P_S, T', F'\rangle$, constructed by applying i and ii, identifies a* weakly connected[6] *TB net, where:*

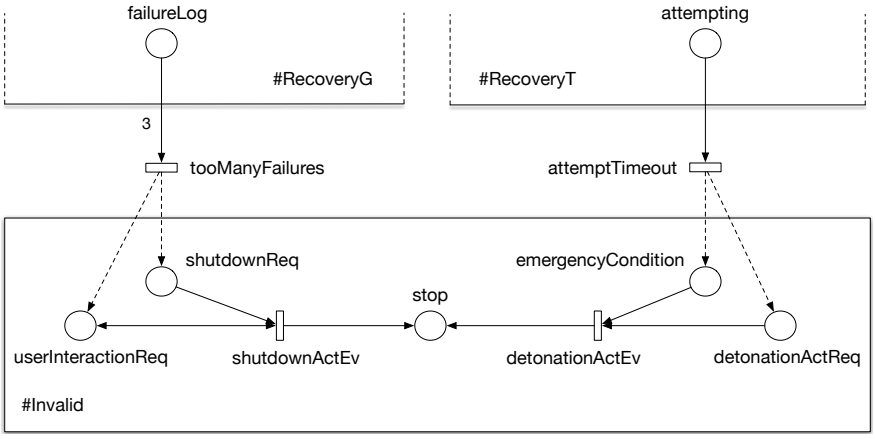
i $T' = T_S \cup {}^\bullet z \cup z^\bullet$

ii $F' = F_S \cup \ \{(t,p) \in F : t \in {}^\bullet z \wedge p \in P_S\} \cup \{(p,t) \in F : p \in P_S \wedge t \in z^\bullet\}$

As an example, consider the TB subnet `#UndesiredG` shown in Figure 7 along with its own preset {`fxPositive`} (with outgoing egdes), and its own postset {`startAntway`, `doNotStart`} (with incoming edges). In this case, we obtain a weakly connected Petri net, therefore the `#UndesiredG` TB net module is a valid zone.

Definition 3. ***Cross-zone transition.*** *Given a TB net $\langle P, T, F\rangle$, a cross-zone transition is a transition $t \in T$ s.t. there exists one and only one zone z, s.t. $t \in z^\bullet$ and there exists at least a zone z' s.t. $t \in {}^\bullet z'$.*

[6] A Petri net is *weakly connected* iff. foreach two elements x and y, there exists an undirected path leading from x to y.



Time-Functions:

`shutdownActEv`	$[\texttt{shutdownReq}, \texttt{shutdownReq} + 1.0]$	`detonationActEv`	$[\texttt{emergencyCondition}, \texttt{emergencyCondition} + 0.5]$
`tooManyFailures`	$[\texttt{failureLog}, \texttt{failureLog} + 0.4]$	`attemptTimeout`	$[\texttt{attempting} + 30, \texttt{max}(\texttt{attempting}, \texttt{askPositionR}) + 30.5]$

Figure 10: The `#Invalid` zone of the UAV system, along with cross-zone transitions. Weak transitions are depicted in gray. The arc labeled with 3 represents a shortcut which stands for 3 arcs with the same source and target.

Considering the UAV case study, the set of all cross-zone transitions is {`fxPositive`, `startAnyway`, `doNotStart`, `startActEv`, `recoveredActEv`, `restrictActEv1`, `restrictActEv2`, `threatActEv`, `idleActEv`, `replanningStart`, `attemptTimeout`, `tooManyFailures`, `threatResActEv`}. The arcs connecting zones to cross-zone transitions are depicted with dashed arrows.

Definition 4. ***Zone-based TB net.*** *A zone-based TB net PN_Z is a TB net $\langle P, T, F \rangle$, composed of a non empty set of zones Z, s.t.:*

i $\forall p \in P,\ \exists! z \in Z : p \in z.$

ii $\forall t \in T,\ \exists! z \in Z : (t \in z\ \vee\ t$ *is cross-zone*$).$

iii $\forall z \in Z, {}^\bullet z \cup z^\bullet \neq \emptyset.$

Back to our running case study, we can individuate seven disjoint zones, introduced early in Figure 6. There are no places outside these zones and transitions are either inside zones or cross-zone. Moreover, all the zones have at least a non empty preset or postset, connecting them to the rest of the system. Therefore, the UAV model follows the definition of zone-based TB net.

A labeling function λ is used to associate a firing transition, representing an action or an event, with a specific zone representing a steady-state behavior.

Definition 5. ***Zone labeling function.*** *Given a zone-based TB net PN_Z, the zone labeling function λ takes as input a transition t and returns a zone z, s.t.:*

$$\lambda(t) = z,\ \textit{iff.}\ t \in z\ \textit{or}\ t \in z^\bullet$$

4.3. Timed Adaptation Models formalization

In our framework, we realize adaptation models by means of *cross-zone* transitions, along with their own temporal functions used to compute temporal constraints upon the dynamically adaptive behavior.

4.3.1. Timed One-point adaptation

The *timed one-point adaptation* is modeled by the pair (t, f_t) composed of a cross-zone transition (either *weak* or *strong*) and its associated temporal function that connects the source zone $z_{\mathcal{S}}$ and the target zone $z_{\mathcal{T}}$, such that places of ${}^\bullet t$ belong to $z_{\mathcal{S}}$ and there exists at least one place in $t^\bullet$ that belongs to $\mathcal{T}$. When the adaptation transition t fires, it performs the transformation between the source and the target zones by consuming the enabling tuple in $z_{\mathcal{S}}$ and producing new tokens in $z_{\mathcal{T}}$. The firing of t can produce tokens into places belonging to multiple zones ($z_{\mathcal{S}}$ itself included). The firing of the transition causes $z_{\mathcal{S}}$ to stop its execution and $z_{\mathcal{T}}$ to start. No other zones are allowed to start their execution, although some places belonging to these zones are marked.

The quiescent states of the zone $z_{\mathcal{S}}$ are those reachable symbolic states that enable a cross-zone transition, while the adaptation set of a timed one-point adaptation transition is composed of a single edge e connecting quiescent states to another region of the state space. Temporal information attached to e (i.e., local minimum-maximum firing times) constitutes the temporal constraint τ.

The temporal semantics associated with the transition t can be either *weak* or *strong*. For instance, the transition between the zones `#Normal` and `#UndesiredG` (see Figure 7) have *weak* semantics because the undesired behavior could happen but it is not forced to. Instead, the transition `replanningStart` between the zones `#UndesiredT` and `#RecoveryT` (see Figure 8) has *strong* semantics because we force the system to adapt itself in order to handle a degraded situation. This transition models a timed one-point adaptation between the undesired zone and the recovery zone. In fact, its firing consumes tokens from places `degradedTrajAlarm` and `pathAdjustReady` (belonging to the source zone) and it produces tokens in place `repairTrajectory` (belonging to the target zone). Once the transition fires, the recovery behavior is the only executable zone.

4.3.2. Timed Overlap adaptation

The *timed overlap adaptation* is modeled by a cross-zone transition (either *weak* or *straong*) connecting the source zone $z_{\mathcal{S}}$ to the target zone $z_{\mathcal{T}}$. The timed overlap adaptation involves the parallel execution of both the source zone and the target zone. The firing of a cross-zone transition consumes tokens in $z_{\mathcal{S}}$ and produces tokens into some places belonging to $z_{\mathcal{S}}$ and some places belonging to $z_{\mathcal{T}}$ (anyway, it is possible to mark also some places belonging to other zones). Once fired, $z_{\mathcal{S}}$ and $z_{\mathcal{T}}$ execute in parallel. No other zones are allowed to start their execution, although some places belonging to these zones are marked. This kind of adaptation is very common in multi-threaded or multi-process programs, where the system spawns a new thread able to deal with the changing execution domain, while another thread finishes its own tasks to reach a consistent state. Moreover, different threads can independently adapt to the target behavior through a one-point adaptation resulting in a overlap adaptation [12].

As an example, consider the cross-zone transition `threatActEv` in Figure 8. It models a timed overlap adaptation between the `#Normal` zone and the `#UndesiredT` zone. Its firing produces tokens into the place `radarShutDownReq` belonging to the source zone and into the places `degradedTrajAlarm` and `threat` belonging to the target zone. Once fired, the source and the target zone execute in parallel, in fact, both the transitions `turnRadarOff` and `retrieveTrajectory` are enabled.

4.3.3. Timed Guided adaptation

In order to define the *timed guided adaptation*, we should create a zone to model the restricted mode. The *restricted zone* behaves similarly to the original zone except that it has some blocked functionalities that allow reaching a quiescent state faster with respect to the original zone. For instance, the restricted zone can be constructed from the original one by removing some transitions that prevent the system to reach a quiescent state. Once defined both the functionalities that should be blocked and the restricted zone, we define a set $\mathcal{G} = \{(t, f_t)\}$, where each t is a cross-zone transition with *weak* semantics. Each transition, along with the temporal function, connects places belonging to the original zone with places belonging to the restricted zone. We can therefore model the possibility for the original behavior to turn into the restricted mode depending on the presence of an adaptation request. It is worth noting that, the larger is $|\mathcal{G}|$, the more is the responsiveness of the system because we increase the number of states able to handle adaptation requests.

Any restricted zone must not violate any global invariant and must not reach deadlock states before reaching a state of quiescence (on due time).

Figure 9 shows this particular form of adaptation in the UAV case study. Whenever an adaptation request caused by a threat happens in a non quiescent state of the `#Normal` behavior. The `#Restricted` zone interrupts the current mission to bring the system into the *flight condition* which represents a quiescent state. At this point the the system adapts itself to restore the nominal functionality through the usual undesired-recovery cycle.

It is worth noting that the *weak* semantics of cross-zone transitions is used to model external uncertainty (i.e., a form of uncertainty arising from the lack of knowledge on the environment or domain in which the software is deployed). For example, external uncertainty for the UAV system may include the unknown presence of bad weather conditions or other threats along the flying path. For example, the cross-zone transition `threatActEv` in Figure 8 has *weak* semantics, meaning that the system can (but it is not forced to) adapt itself to handle a degraded situation, depending on the evaluation performed at runtime on the data retrieved from the radar component.

4.4. State space construction

Once all the zones in the TB net model of the system are created and connected by means of cross-zone transitions, we can construct the state space (i.e., the TRG) and then mechanically identify regions and adaptation sets. During the state space exploration we associate reachable symbolic states with steady-state behaviors. Moreover, we map timed adaptation models into adaptation sets following specific rules depending on the adaptation type.

In order to identify the regions, we formally introduce the concept of *active zone mapping*:

Definition 6. ***Active Zone Mapping****. Given a zone-based TB net PN_Z, and the structure $TRG(PN_Z) = \langle N, E, S_0 \rangle$, the active zone mapping Λ is a function that accepts as input a reachable state S and it returns a set of zones in Z, s.t.:*

$$\Lambda(S) = \{z : \exists \langle S, t, S' \rangle \in E \text{ s.t. } \lambda(t) = z\}$$

The rationale of this function is to use firing transitions to identify the current behavior of the system. In particular, we identify the steady-state behavior associated to a reachable state by looking at the zone responsible of an occurring event. For instance, given the following initial state of the UAV example:

$$S_0 = \langle \texttt{idle}\{T_0\}, \texttt{checkpoint}\{T_0\};\ T_0 \geq 0 \wedge T_0 \leq 10 \rangle,$$

the evaluation of $\Lambda(S_0)$ returns a set of zones containing only the zone `#Normal`. In fact the only enabled transitions in S_0 is `missionStart` that maps into the normal behavior zone. Indeed, it represents a normal event.

The concept of *region*, informally introduced in section 2.3 (Figure 4), is formally defined as follows:

Definition 7. ***Region****. Given a zone-based TB net model PN_Z, its state space $TRG(PN_Z)$, and the active zone mapping function Λ, a region is a set of reachable states $\mathcal{H}$, s.t.:*

i $\forall S, S' \in \mathcal{H} : S \neq S'$, $\Lambda(S) = \Lambda(S')$ *and* $\forall z \in \Lambda(S)$, z *is not restricted*

ii $\mathcal{H}$ *locates in* $TRG(PN_Z)$ *a weakly connected*[7] *component.*

The concept of *adaptation set* (informally introduced in section 2.3) can be formally characterized using the definitions of *active zone mapping function* and *region*.

[7] A component is *weakly connected* if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.

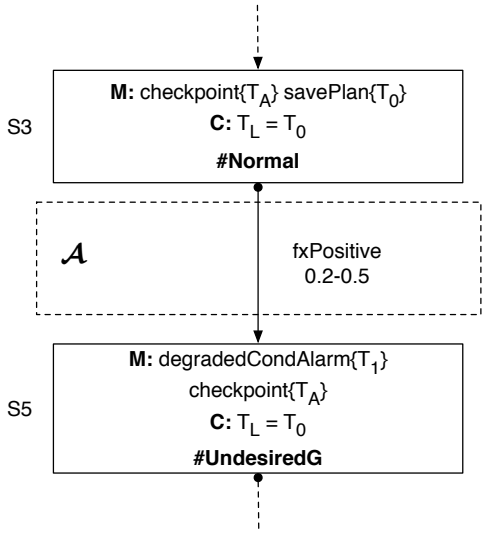


Figure 11: Example of *timed one-point adaptation set* between the `#Normal` region and the `#UndesiredG` region.

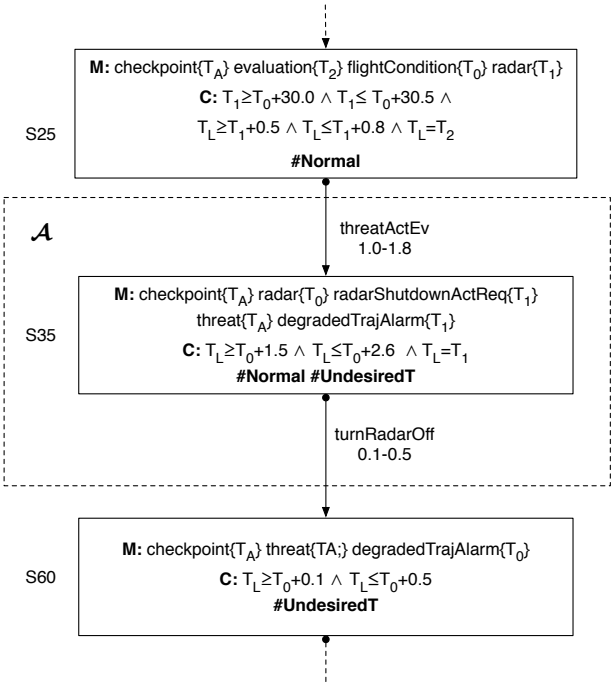


Figure 12: Example of *timed overlap adaptation set* between the `#Normal` region and the `#UndesiredT` region.

Definition 8. ***Timed Adaptation Set.*** *Given a zone-based TB net PN_Z, the state space $TRG(PN_Z)$ and two different regions $\mathcal{S}$ and $\mathcal{T}$, a timed adaptation set $\mathcal{A}$ is a set of weakly connected $TRG(PN_Z)$ elements (states and edges) connecting $\mathcal{S}$ to $\mathcal{T}$.*

Definition 9. ***Timed One-point Adaptation Set.*** *A timed one-point adaptation set is a timed adaptation set $\mathcal{A}$ s.t. $\mathcal{A}$ contains a single edge labeled with a cross-zone transition connecting the two different regions $\mathcal{S}$ and $\mathcal{T}$.*

Definition 10. ***Timed Overlap Adaptation Set.*** *A timed overlap adaptation set is a timed adaptation set composed of a set of states $\mathcal{H}$ and a set of edges $\mathcal{I}$ connecting the two different regions $\mathcal{S}$ and $\mathcal{T}$, s.t.:*

i $\forall S \in \mathcal{H}$, $|\Lambda(S)| > 1$

ii *$\mathcal{I}$ contains all the outgoing edges departing from states in $\mathcal{H}$, and the incoming edges of states in $\mathcal{H}$ departing from states outside $\mathcal{H}$.*

Definition 11. ***Timed Guided Adaptation Set.*** *A timed guided adaptation set is a timed adaptation set composed of a set of states $\mathcal{H}$ and a set of edges $\mathcal{I}$ connecting the two different regions $\mathcal{S}$ and $\mathcal{T}$, s.t.:*

i $\forall S \in \mathcal{H}$, $\forall z \in \Lambda(S)$, *z is a restricted zone*

ii *$\mathcal{I}$ contains all the outgoing edges departing from states in $\mathcal{H}$, and the incoming edges of states in $\mathcal{H}$ departing from states outside $\mathcal{H}$.*

Figure 11 shows an example of timed one-point adaptation set. Two disjoint regions reifying the `#Normal` and the `#UndesiredG` behaviors, respectively, are connected by a single edge labeled with the firing transition and temporal information. The zone names upon states are the results of the evaluation of the Λ function.

The adaptation set $\mathcal{A}$ of a timed overlap adaptation transition is composed of a set of states and edges connecting quiescent states of $\mathcal{S}$ to states of $\mathcal{T}$. Each state in $\mathcal{A}$ maps to both the source and the target regions. The temporal constraint τ is identified by the set of temporal information attached to the edges in $\mathcal{A}$. Figure 12 shows an example of timed overlap adaptation set. The adaptation set is composed of the set of states $\{S_{35}\}$ along with the proper edges labeled with the firing transitions and temporal information. Symbolic states in the adaptation set $\mathcal{A}$ belong to the normal behavior and undesired behavior regions at the same time, therefore the system exhibits multiple steady-state behaviors.

Finally, Figure 13 shows an example of timed guided adaptation set. The adaptation set is composed of the set of states $\{S_{56}, S_{82}, S_{105},\}$ that map through the Λ function into a restricted zone, along with the proper edges labeled with the firing transitions and temporal information. The outgoing boundaries of $\mathcal{A}$ are made up by a one-point adaptation leading from $\mathcal{A}$ into the target region (i.e., the `#UndesiredT` region).

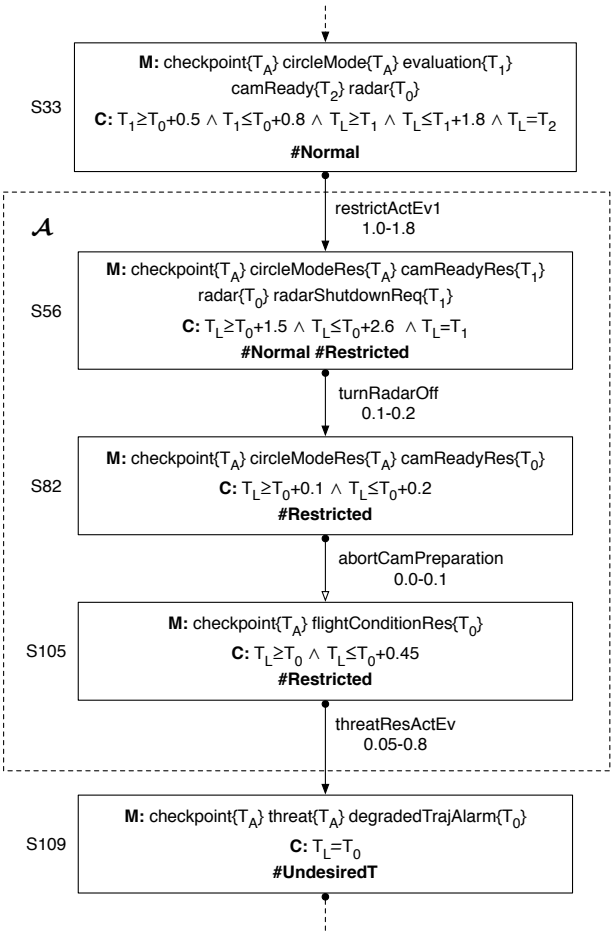


Figure 13: Example of *timed guided adaptation set* between the `#Normal` region and the `#UndesiredT` region.

5. Formal verification

Based on the specification formalism described in the previous Section, we are able to verify the correctness of real-time self-adaptive systems by essentially inspecting the *TRG*. In this Section, we describe all types of properties, *structural* and *behavioral*, supported by our verification approach. This Section is divided into two main parts. Subsection 5.1 introduces the syntax and the semantics of all the available

properties, along with examples of property verification from the UAV case study. Subsection 5.2 introduces the main verification algorithms along with their time and space complexity evaluation.

5.1. Verifiable properties

Our framework supports the verification of three categories of properties: *structural properties*, *adaptation meta-properties* and *system behavioral properties*.

5.1.1. Structural properties

These properties aim at validating a zone-based TB net model w.r.t. the expected subdivision in zones. Indeed, each zone must satisfy the structural properties introduced in Definition 2, Section 4. The syntax accepted by our software tool is:

```
$ zonecheck(#z)
```

where z is the name of a zone of the considered TB net model.

Example of properties verified for the zones of the UAV model are reported below (SP1-SP4).

```
$ zonecheck(#Normal)        SP1
$ zonecheck(#UndesiredT)    SP2
$ zonecheck(#RecoveryG)     SP3
$ zonecheck(#Invalid)       SP4
```

If the property holds for the target zone, the software tool returns a positive answer and lists both the incoming and the outgoing cross-zone transitions.

5.1.2. Adaptation meta-properties

These properties do not depend on the specific modeled system. They are general properties related to adaptation and any self-* system (self-healing in our case) should satisfy them. They are divided into four categories: cross-zone transitions meta-properties, safety meta-properties, robustness meta-properties, timed meta-properties.

• *Cross-zone transition meta-properties.* They aim at checking the conformance of the generated *adaptation sets* w.r.t. the intended adaptation models. For each cross-zone transition t of the model, t must connect two disjoint regions z and z' with the proper adaptation set, depending on the specific employed timed adaptation model. The syntax accepted by our software tool is:

```
$ onepoint(#z, t, #z')
$ overlap(#z, t, #z')
$ guided(#z, t, #z')
```

where z and z' are the names of zones and t the name of a cross-zone transition of the considered TB net model. These commands, one for each adaptation model, are used to verify the characterizing properties of the adaptation models as described in section 4.3. For instance, the *onepoint* meta-property checks if the regions z and z' are connected through a single edge labeled with t. Moreover, they prove the *adaptation integrity constraint*.

Some cross-zone properties verified upon the UAV model are reported below (AP1-AP3).

```
$ onepoint(#Normal, fxPositive, #UndesiredG)          AP1
$ guided(#Normal, restrictActEv1, #Restricted)        AP2
$ overlap(#Normal, threatActEv, #UndesiredT)          AP3
```

• *Safety meta-properties.* These properties are used to verify whether the system can reach an invalid state (i.e., either a deadlock state or a state that belongs to the invalid behavior region). The syntax of the property is as follows:

```
$ !(#z ?> #Invalid)
```

where "a ?> b" is used to verify that "whenever a holds, there exists a path leading to b". An example of safety meta-property, verified upon the UAV model, is the following:

$!(#Normal ?> #Invalid) **AP4**
FALSE
S0(#Normal) [0.0,100.0] S1(#Normal) [0.1−0.5] ... S173(#RecoveryT) [0.0,0.05] S176(#Invalid)

Since the property does not hold ($FALSE$), the verification framework returns a feasible path as a counterexample. The path contains information about regions associated with states and temporal information attached to edges (local minimum-maximum firing times).

An important observation, emerging from the evaluation of property AP4, is that the invalid behavior cannot be completely avoided. In fact, the re-planning task modeled in the `#RecoveryT` zone calculates a new trajectory in order to take a detour around the threats, if possible. This means that the constraints imposed by multiple threats areas could make this computation unfeasible, thus leading the planning algorithm to a failure. This eventuality is modeled by the weak transition `planningFailR` that can (but is not forced to) fire before the `planningOkR` transition.

• *Robustness meta-properties.* Robustness properties are specific reachability properties used to verify that the system is able to recover from a failure. These properties must be verified to ensure self-healing capability of the system. The following robustness meta-properties (AP5-AP7) were verified upon the UAV model:

$ #Normal ?> #UndesiredT **AP5**
$ #UndesiredT −> #RecoveryT **AP6**
$ #RecoveryT ?> #Normal **AP7**

Note that the property "a –> b" is used to verify that "whenever a holds, eventually b will happen". The three properties examples reported above prove that there exists a path connecting a normal state to an undesired state (AP5); for each undesired state, we always reach a recovery state (AP6); for each recovery state, is always possible to reach a normal state (AP7).

The evaluation of the following properties (AP8-AP11) proves that the undesired behavior is reachable from the normal behavior either directly or passing through the restricted mode.

$ #Normal ?> #UndesiredG **AP8**
TRUE
S0(#Normal) [0.0,100.0] S1(#Normal) [0.1,0.5] S2(#Normal) [0.0,1.2] S3(#Normal) [0.2,0.5] S5(#UndesiredG)

$ #Normal ?> #UndesiredT **AP9**
TRUE
S0(#Normal) [0.0,100.0] S1(#Normal) [0.1,0.5] ... S84(#Normal) [0.0,0.0] S89(#UndesiredT)

$ #Normal ?> #Restricted **AP10**
TRUE
S0(#Normal) [0.0,100.0] S1(#Normal) [0.1,0.5] S65(#Normal) [1.0,1.8] ... S78(#Normal, #Restricted)

$ #Restricted −> #UndesiredT **AP11**
TRUE

The evaluation of the properties AP12-AP13 proves that once the system reaches an undesired behavior, it always adapts itself to work in recovery mode.

$ #UndesiredG −> #RecoveryG **AP12**
TRUE

$ #UndesiredT −> #RecoveryT **AP13**
TRUE

Once the recovery behavior is accessed, it is possible to restore the normal functionality. This is proven by the following properties (AP14-AP15):

$ #RecoveryG ?> #Normal **AP14**
TRUE
S10(#RecoveryG) [0.1,0.5] S0(#Normal)

$ #RecoveryT ?> #Normal **AP15**
TRUE
S86(#RecoveryT) [1.2,1.5] S90(#RecoveryT) [0.1,0.5] S93(#Normal)

• *Timed meta-properties.* The previous properties can be extended by adding temporal constraints. This is particularly useful to check whether the adaptation behavior is carried out within a specific temporal deadline t, i.e., there exists a feasible path from the source zone to the target zone, taking at most t time units. For instance, the properties AP16 and AP17 have been verified in order to assess the time required by the system to restore the normal functionality from different undesired zones.

$ *#UndesiredG ?> #Normal, 0.8* **AP16**
TRUE
S7(*#UndesiredG) [0.5,100.0] S11(#RecoveryG) [0.2,0.5] S14(#RecoveryG) [0.1−0.5] S4(#Normal)*

$ *#UndesiredT ?> #Normal, 1.8* **AP17**
TRUE
S70(*#UndesiredT) [0.5,0.8] S86(#RecoveryT) [1.2,1.5] S90(#RecoveryT) [0.1,0.5] S93(#Normal)*

Whenever the property holds, the tool supplies a feasible path as a proof of correctness.
Another interesting timed property verified upon the UAV is (AP18):

$ *#UndesiredT −> #RecoveryT, 2.1* **AP18**
TRUE

This allows us to prove the maximum response-time bound of the system in a particular degraded situation, i.e., the `#UndesiredT` zone, in charge of recognizing and evaluating an occurring threat.

5.1.3. System behavioral properties

These properties go into finer detail and require specific knowledge on the system to analyze. They are divided into *intra-zone properties* and *inter-zone properties*.

Intra-zone properties. These properties aim at verifying the correctness of zones in isolation. In particular, we can verify *invariant*, *safety*, and *liveness* properties. The properties in this category can be verified by inspecting the *TRG* computed only on the zones of interest.

• *Invariant properties.* Invariants should be preserved for each reachable state of the zone. We can express invariants by means of a boolean combination of conditions on the number of tokens in places. For instance, the BP1 invariant has been verified upon the `#Normal` zone in order to prove that for each reachable state (operator A), the `radar` place is marked whenever at least one place among `flightCondition`, `circleMode` and `landingPosition` is marked (indeed, for each reachable state of the normal zone, the radar is active when the vehicle is either in flight condition, circle mode or landing position).

$ A(radar!=1 || (flightCondition==1 || circleMode==1 || landingPosition==1)) **BP1**

Similar invariants have been proven also for the `askTrajectory` (BP2) and the `saveDestination` (BP3) places.

$ A(askTrajectory!=1 || (takeoff==1 || climbing==1 || maneuver==1)) **BP2**
$ A(saveDestination!=1 || (takeoff==1 || climbing==1 || maneuver==1)) **BP3**

Other examples of invariants involving other zones are reported below (BP4, BP5).

$ A(checkpoint==1 || waiting==1) **BP4**
$ A(degradedTrajAlarm!=1 || (pathAdjustReady==1 || threat==1)) **BP5**

The BP4 invariant verifies that for each reachable state of the `#UndesiredG` zone, either the `checkpoint` place or the `waiting` place is marked. The BP5 invariant is used to prove that for each reachable state of the `#UndesiredT` zone, if the trajectory alarm is active, then a threat has just been recognized or the system is ready to start the recovery procedure.

• *Safety properties.* Safety properties are used to verify that "something bad will never happen". Properties BP6 and BP7 are examples of intra-zone safety properties verified on the `#Normal` behavior zone.

$!E(dataReady>1) **BP6**
$!E(evaluation>1) **BP7**

These properties are used to prove that it does not exist (E operator) a reachable state of the normal behavior zone such that unprocessed data produced by the radar component is accumulated during the threat assessment procedure.

Additional examples of intra-zone safety properties are reported below (BP8-BP10).

```
$ !E(degradedCondAlarm==1 && showTrajectory==0)                    BP8
$ !E(planModification==1 && planErasure==1)                        BP9
$ !E(attempting==1 && savePlanR==1)                                BP10
```

The BP8 property is used to verify that it does not exist a reachable state of the `#UndesiredG` zone such that the alarm is active and the current trajectory is not displayed to the system user. The BP9 property verifies that the `#RecoveryG` zone does not reach a state where the plan is both modified and erased. The BP10 property is used to prove that in the `#RecoveryT` zone it is not possible to compute a new trajectory and save the plan at the same time. In fact, the latter operation can be performed only after the planning algorithm succeeded, therefore the `attempting` place must be empty.

• *Liveness properties.* We can verify two different types of liveness properties characterized by the two different operators "a –> b" and "a ?> b" (already introduced in section 5.1.2), also including temporal constraints.

Some examples of (timed) liveness properties verified upon the `#Normal` zone in isolation are reported below (BP11-BP13).

```
$ saveTrajectory==1 −> takeoff==1, 1.5                             BP11
   TRUE

$ saveTrajectory==1 −> climbing==1, 3.8                            BP12
   TRUE

$ saveTrajectory==1 ?> (flightCondition==1 && radar==1), 12.8       BP13
   TRUE
   S4(#Normal) [1.5,2.0] S6(#Normal) [0.5,0.5] S8(#Normal) [0.0,0.0] S13(#Normal) [10.0−100.0] S16(#Normal)
   [0.8,1.2] S17(#Normal)
```

The BP11 property is used to verify that is always possible to reach the takeoff phase once the trajectory has been computed within 1.5 time units. The BP12 property is used to verify that is always possible to reach the climbing phase once the trajectory has been computed within 3.8 time units. The BP13 property is used to prove that the flight condition, along with the radar activation, is reachable within 12.8 time units from the trajectory computation.

Other examples of similar liveness properties verified upon the `#Restricted` and the `#RecoveryT` zones, respectively, are reported below (BP14, BP15).

```
$ circleModeRes==1 −> circleModeRes==0 && flightConditionRes==1, 0.31   BP14
   TRUE

$ attempting==1 ?> savePlanR==1, 1.7                               BP15
   TRUE
   S94(#RecoveryT) [1.2,1.5] S97(#RecoveryT) [0.5,0.8] S90(#RecoveryT)
```

Inter-zone properties. These properties aim at verifying the correctness of the behavior of the entire system. In particular we can verify interesting *invariant*, *safety*, and *liveness* properties on the *TRG* built from the whole system model.

• *Invariant properties.* Two examples of inter-zone invariant properties are reported below (BP16-BP17). They are used to verify the proper mutual exclusion between two specific marked places representing the normal and the restricted mode behaviors, respectively.

```
$ A(circleModeRes!=1 || circleMode==0)                             BP16
$ A(circleMode!=1 || circleModeRes==0)                             BP17
```

Indeed, the `circleMode` place should never be marked at the same time with `circleModeRes`.

• *Safety properties.* The properties BP18 and BP19 represent two examples of inter-zone safety properties.

$!E(degradedCondAlarm==1 && recoveryGAlarm==1) **BP18**
$!E(degradedCondAlarm==1 && degradedTrajAlarm==1) **BP19**

The BP18 property is used to verify that it does not exist a reachable state where the `#UndesiredG` and `#RecoveryG` behaviors overlap. In fact, the former zone adapts into the latter one through a one-point adaptation, therefore the alarm cannot be activated by the two zones at the same time. The BP19 property verifies that it does not exist a reachable state where the two different alarms (i.e., the `#UndesiredG` and the `#UndesiredT` alarms) are active at the same time.

• *Liveness properties.* Inter-zone liveness properties can be verified similarly to the intra-zone ones, but they involve conditions defined upon multiple zones. Some examples of (timed) inter-zone liveness properties follow below (BP20, BP21).

$ radar==1 && threat==1 −> radar==0, 0.1 **BP20**
TRUE

$ evaluation==1 && flightCondition==0 ?> trajAlarm==1 && threat==1, 1.1 **BP21**
TRUE
S29(*#Normal) [1.0,1.8] S48(#Restricted, #Normal) [0.0,0.2] S67(#Restricted, #Normal) [0.0,0.5] S85(#Restricted)* [0.0,0.8] S89(*#UndesiredT)*

The BP20 property is used to verify that once a threat has been accessed, the radar component is turned off within 0.1 time units. The BP21 property proves that the threat assessment during the flight condition can lead to a threat alarm after 1.1 time units.

5.2. Algorithms and complexity evaluation

In the following we report the main verification algorithms divided for each category along with time and space complexity evaluation.

5.2.1. Structural properties

The implementation of the `zonecheck` procedure uses the Tarjan's algorithm [45] to find the weakly connected components of the zone-based TB net model with structure $\langle P, T, F\rangle$. Therefore, both the worst case time complexity and space complexity of this procedure are $\mathcal{O}(|P + T + F|)$, i.e., linear in the size of the TB net structure.

5.2.2. Adaptation meta-properties

• *Cross-zone transition meta-properties.* The software implementation evaluates these properties (e.g., AP1-AP3) through a depth first exploration procedure of the $TRG = \langle N, E, S_0\rangle$ to verify the structure of the target adaptation set (section 4.3). Therefore, the worst case time complexity and space complexity of this procedure are $\mathcal{O}(|N + E|)$ and $\mathcal{O}(|N|)$, respectively.

• *Safety, Robustness and Timed meta-properties.* *Safety* properties (e.g., AP6-AP7) and *robustness* properties (e.g., AP8-AP15), that make use of the "?>" and the "–>" operators, correspond to specific Computation Tree Logic (CTL) [46] formulas. In particular, the semantics of the following formulas:

$$\phi \ ?> \psi \qquad (1) \qquad\qquad \phi \ -> \psi \qquad (2)$$

can be expressed by using CTL operators respectively as follows:

$$AG(\phi \rightarrow EF\psi) \qquad (3) \qquad\qquad AG(\phi \rightarrow AF\psi). \qquad (4)$$

Intuitively, the formula (3) means that foreach path (i.e., the universal path operator A), globally (i.e., the path-specific operator G), whenever ϕ holds, there exists a path (i.e., the existential path operator E) where ψ eventually (i.e., the path-specific operator F) holds. The formula (4) mans that foreach path, globally, whenever ϕ holds, ψ eventually holds foreach path. Thus, (3) and (4) are semantically equivalent to (1) and (2), respectively.

Timed meta-properties (e.g., AP16-AP18) are used to write bounded-response time properties, such as:

$$\phi \ ?> \psi,\ t \qquad (5) \qquad\qquad \phi \ -> \psi,\ t'. \qquad (6)$$

In this case, the semantics of (5) and (6) can be expressed by means of specific formulas in Timed CTL, or simply TCTL [47] (i.e., an extension of the classical untimed branching time logic CTL with time constraints), respectively as follows:

$$AG(\phi \rightarrow EF_{\leq t}\psi) \qquad (7) \qquad\qquad AG(\phi \rightarrow AF_{\leq t'}\psi). \qquad (8)$$

The meaning of (7) and (8) is similar to *safety* and *robustness* formulas, respectively, except for the time constraints t and t' in $\mathbb{R}_{\geq 0}$ on the path-specific operator F.

The software tool verifies *safety*, *robustness* and *timed* properties by means of classic TCTL model checking procedures [48, 49] able to work on the *TRG* structure. The model checking problem for TCTL is PSPACE-Complete [47]. In particular, the worst case time complexity of these procedures is $\mathcal{O}(|\Phi|\cdot|N+E|)$, i.e., linear in the size of the Φ formula (due to the recursive descent over the parse tree of Φ) and in the size of the $\langle N, E, S_0\rangle$ structure of the *TRG*. The worst case space complexity is $\mathcal{O}(|N+E|)$, which is the memory required to maintain the entire *TRG* structure.

5.2.3. System behavioral properties

The software tool verifies the properties in this category by means of classic TCTL model checking procedures able to work on the TRG structure. Therefore, time and space complexity analysis follow the observations reported above.

• *Invariant and Safety properties.* Like the *safety* adaptation meta-properties, both *intra-* and *inter-*zone *invariant* properties (e.g., BP1-BP10) and *safety* properties (e.g., BP16-BP19), correspond to specific CTL formulas. In particular, the syntax of these two types of property are reported below:

$$A(\phi) \qquad (9) \qquad\qquad !E(\psi). \qquad (10)$$

The semantics of (9) and (10) can be expressed by using CTL operators, respectively, as follows:

$$AG(\phi) \qquad (11) \qquad\qquad \neg EF(\phi). \qquad (12)$$

The formula reported in (11) means that foreach path, globally, the proposition ϕ (i.e, a Boolean combination of conditions on the number of tokens in places) holds. The formula reported in (12) mans that does not exist a path, where the proposition ϕ eventually holds. Thus, (11) and (12) are semantically equivalent to (9) and (10), respectively.

• *Liveness properties.* Like the *robustness* and *timed* adaptation meta-properties, both *intra-* zone *liveness* properties (e.g., properties BP1-BP10) and *inter-*zone *liveness* properties (e.g., properties BP16-BP19) make use of the bounded-response time "–>" and "?>" operators, therefore they correspond to specific TCTL formulas. Similarly to *invariant* and *safety* behavioral properties, both the sub-formulas ϕ and ψ occurring in (7) and (8) are propositions composed of a set of Boolean conditions on the number of tokens in places.

6. The ZAFETY software tool

The formal framewotk described in this article has been implemented as a software tool set written in Java, so called ZAFETY[8]. This tool extends Graphgen [32], in order to support both the specification phase, through a graphic editor, and the verification phase, through the exploration of regions and adaptation sets of the *TRG* structure.

[8]The source code and the executable binaries of the ZAFETY editor and the ZAFETY engine, along with runnable examples can be found at: `http://camilli.di.unimi.it/zafety`.

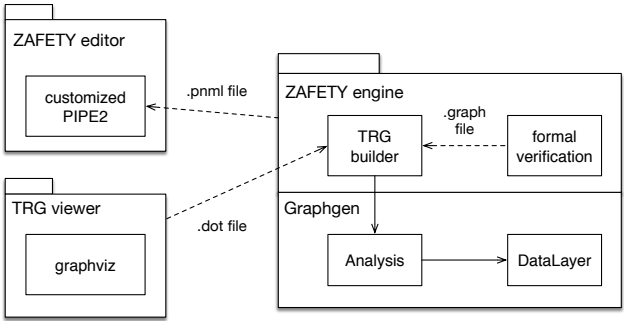


Figure 14: The main components of ZAFETY.

The tool architecture (see Figure 14) includes different interconnected components that communicate by means of proper files.

The ZAFETY editor is a customized version of the PIPE2 open source tool [50]. It supplies a graphical user interface to create and edit arbitrary complex zone-based TB net models. A drawing toolbar (including places, transitions, arcs and tokens) allows the user to draw a model into the main canvas through simple drag and drop gestures (see Figure 15).

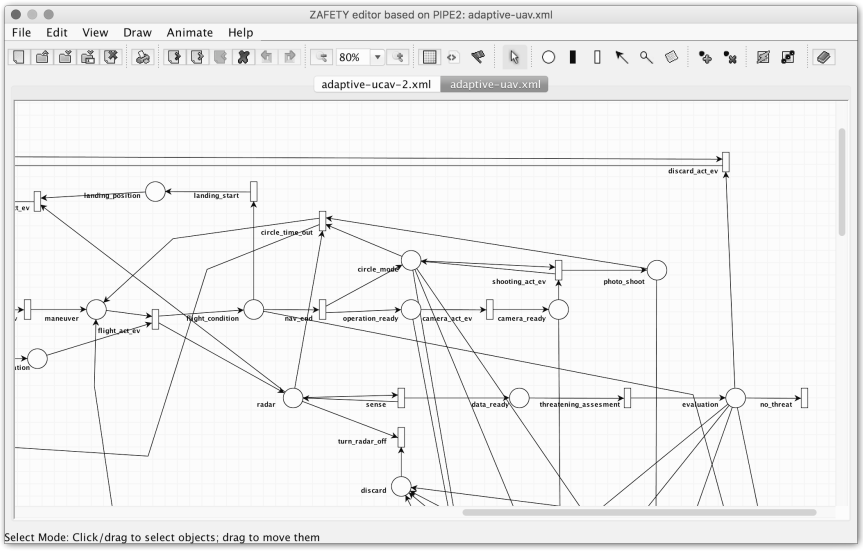


Figure 15: The main window of the ZAFETY zone-based TB nets editor.

Figure 16 shows the place editor window. As we can see, it allows the item to be annotated with the place name, the zone name, the number of tokens, and the variables representing the symbolic timestamps associated with tokens. Figure 17 shows the transition editor window. In this case it is possible to change the transition name, the name of the zone, the parametric time interval bounds *tmin* (lb_t), *tmax* (ub_t), and the temporal semantics. The user interface of the editor also permits to export a graphical view (as PNG or PostScript file) and to save the PNML [51] representation of the model (as a XML file) for further analysis through the ZAFETY engine.

The ZAFETY engine has been implemented on top of GRAPHGEN [32], a JAVA software tool for the reachability analysis of TB nets. In fact, the *TRG builder* exploits the functionality of the *Analysis* and the *DataLayer* modules of GRAPHGEN in order to create the reachable symbolic states of a TB net model.

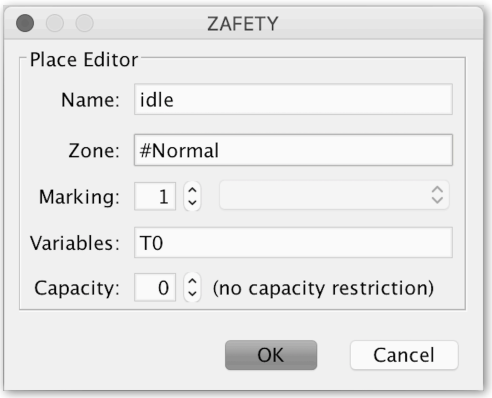


Figure 16: The place editor window of the ZAFETY zone-based TB nets editor.

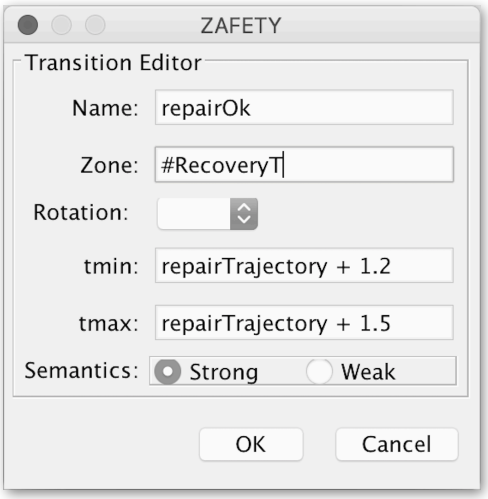


Figure 17: The transition editor window of the ZAFETY zone-based TB nets editor.

Given a PNML file, it generates as output the TRG (enriched with regions and adaptation sets) in binary format that can be used in turn by the *formal verification* module. The tool can also create an annotated DOT text format, used by the GraphViz software tool [52] to generate a graphical visualization of the TRG.

The *formal verification* module can be used to verify a number of correctness properties (introduced in section 5). It grants the possibility of verifying specific requirements upon adaptation through by means of *cross-zone transition properties*. Moreover, it supports the verification of *timed properties* in order to check real-time constraints, and it supports the verification of *robustness properties* to verify time-constrained self-healing behaviors.

6.1. Performance evaluation

In order to evaluate the performance of the *TRG builder* and the *formal verification* components of the ZAFETY tool, we report in the following some measurements performed during the verification of two case studies: the self-adaptive UAV system (i.e., the running example of this article) and the self-adaptive Gas Burner (GB) system (i.e., a smaller example introduced in our previous work [15]). We ran our experiments on a machine equipped with a Intel Xeon E5-2630 at 2.30GHz CPU, 32GB of RAM, the Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-39-generic x86_64) operating system, and the Java HotSpot 1.8 64-Bit Server virtual machine.

Table 2 reports the performance of the *TRG builder* component used to analyze our case studies. In particular, we report the model size in terms of number of places $|P|$ and number of transitions $|T|$; the *TRG* size in terms of number of *created states* and number of *erased states* (i.e., those states found equals or included into other states during the *TRG* building); the execution time in seconds; and the average memory consumption in Kylobytes. The GB is a simple model and generates a small *TRG* (31 symbolic states). This computation takes approximately 1 second and the average memory consumption is approximately 28 Megabytes. The UAV is instead a more complex system that generates a *TRG* of 2096 symbolic states. This computation takes approximately 2 minutes and the average memory consumption is approximately 240 Megabytes.

Table 3 reports the performance of the *formal verification* component. For each type of property we report the worst case time and space complexity (discussed in section 5), the average execution time (in seconds) and the average memory consumption (in Kylobytes) during the verification of a number of correctness properties on each case study.

case study	|P|, |T|	created states	erased states	exec. time (s)	memory (KB)
GB	26, 20	49	18	1,041	28846
UAV	49, 52	6754	4658	130,189	241321

Table 2: Evaluation of the *TRG builder* component.

Clearly, both the average execution time and the average memory consumption, is higher in the UAV case study for each category, because of the larger *TRG* size and the larger model size. The structural properties usually take less time and consume less memory, in fact both the time and space complexity depend on the model size instead of the *TRG* size. Intra-zone properties usually need less resources (both time and memory) than the inter-zone ones. In fact, although they have the same worst case time complexity and space complexity, they search in a single region, which is usually far smaller than the entire *TRG*. However, the average execution time needed to verify the properties of each category, is very small in both the case studies (few milliseconds). Moreover the average memory consumption is negligible (few Kylobytes).

			GB		UAV	
Properties	time complexity	space complexity	execution time (s)	memory (KB)	execution time (s)	memory (KB)
structural	$\mathcal{O}(\|P+T+F\|)$	$\mathcal{O}(\|P+T+F\|)$	0,014	5667	0,017	6334
Cross-zone transition	$\mathcal{O}(\|\text{N+E}\|)$	$\mathcal{O}(\|N\|)$	0,018	6204	0,039	8760
Robustness	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,019	7741	0,038	9522
Timed	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,022	8125	0,063	11566
Intra-zone Invariants	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,010	8334	0,055	10334
Intra-zone safety	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,008	8628	0,027	12182
Intra-zone liveness	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,006	8360	0,020	13786
Inter-zone Invariants	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,045	15197	0,081	16148
Inter-zone safety	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,031	13510	0,073	15182
Inter-zone liveness	$\mathcal{O}(\|\Phi\| \cdot \|N+E\|)$	$\mathcal{O}(\|N+E\|)$	0,019	9380	0,054	16786

Table 3: Evaluation of the *formal verification* component.

7. Comparison with the state of the art

SA has been widely studied in the software architecture community [53]. Various mechanisms and frameworks for handling adaptation have been proposed, such as (to name a few): SA with aspect-orientation, Dynamic Reconfiguration, Model-Driven Development frameworks for SA, and frameworks for self-optimization (including the adaptation cost itself) [1, 2, 54, 55, 56, 57].

Here we focus on the main formal frameworks for the design and analysis of SA with time constraints. In Subsection 7.1, we review the main approaches existing in the literature, while in Subsection 7.2 we briefly compare them (including our framework) with respect to generally agreed key features.

7.1. Related work

The framework presented in this article has been influenced by some previous works on formal specification and verification of self-adaptive systems with time constraints. In particular, we focus on those approaches adopting model checking as main verification technique.

In [13], J. Zhang and B. Cheng propose a modeling approach based on Petri Nets and a model checking technique based on LTL (*Linear Temporal Logic*) to verify correctness of adaptation and robustness properties. According to specific adaptation models – *one-point adaptation*, *overlap adaptation*, and *guided adaptation* – a set of adaptations realize transitions among steady-state programs in response to environmental changes.

Another Petri Net-based approach, called *Learning Petri Nets*, has been recently presented in [58] to model and verify self-adaptive systems. This formalism is proposed as an extension of hybrid Petri nets that embed some special transitions with neural network algorithms. This formalism can describe both discrete states of the system behavior and continuous states of environment changes through continuous transitions, which generate environmental values at run time. Their models are adaptive through the learning ability of neural network and can be used to optimize decision making at run-time when the run-time environment data is available. However, the proposed approach does not deal with verification of time properties.

In [59], Timed Automata are used to model a decentralized Traffic Monitoring self-adaptive system by adopting the MAPE-K control loop model [3, 60] for autonomic and self-adaptive software systems. A network of Timed Automata allows specifying the behavior of MAPE-K components that synchronize through clock variables and interact via channels. The Uppaal model checker is used to verify flexibility and robustness properties expressed in timed computation tree logic (TCTL) – a computational tree logic extended with clock variables. In [61], D. G. de la Iglesia and D. Weyns adopt the Timed Automata-based formalization approach described above to realize the self-adaptive layer of a Mobile-Learning Application in order to make the application more robust w.r.t. insufficient GPS accuracy. The Uppaal's TCTL was used to specify and verify four groups of properties: functional correctness, GPS service adaptation, self-healing, and MAPE loops interference.

The framework ActivFORMS (Active FORmal Models for Self adaptation) [62] allows continuous verification of adaptation goals at run-time as well as dynamic updates of the formal models in order to support unanticipated changes and assure that the adaptation goals verified at design-time are guaranteed also at run-time. The framework allows modeling MAPE-K feedback loops explicitly in terms of networks of Timed Automata [63]. The approach is evaluated with a small scale system in which robots perform transportation tasks in a warehouse environment. The model of the feedback loop is executed by a virtual machine. The virtual machine has an internal clock that is incremented by time steps. For each time step, the virtual machine identifies the enabled node for each automaton and checks whether the time step would invalidate the time invariants of the enabled nodes. The virtual machine then execute tasks associated with these invalidated nodes in non-deterministic order. The Uppaal model checker is used to verify flexibility and robustness properties expressed in TCTL. This approach ensures at run-time that adaptation behavior starts on due time, but it does not supply a means to verify that correct actions complete within specific deadlines.

MechatronicUML [64] addresses the challenge of designing embedded real-time self-optimizing systems by proposing a coherent and integrated model-driven development approach which supports the modeling and verification of safety guarantees for systems with reconfiguration of software components at run-time. Modeling is based on a syntactically and semantically rigorously defined and partially refined subset of the UML. Verification is based on a special type of decomposition and compositional model checking to make it scalable.

In [65], the SA-CIRCA (self adaptive cooperative intelligent real-time control architecture) architecture for intelligent autonomous systems is proposed. SA-CIRCA automatically synthesizes its control programs or plans from primitive descriptions of the system it is controlling, the system objectives, and the environment in which the system operates. The synthesis process itself can be managed to conform to real-time deadlines that may constrain the time available for reconfiguration.

M. Zeller and C. Prehofer in [66] address the problem of run-time adaptation in networked embedded systems with tight real-time constraints. For such systems, they aim at adapting the placement of software components on networked hardware components at run-time without violating real-time constraints. An adaptation process is formalized in terms of a state transition system with time constraints. Then, they adopt and compare two approaches for finding solutions in the resulting search space for adaptations. One approach is based on planning algorithms and the other one is based on constraint solving. The planning-based approach starts from the current configuration and aims at finding a sequence of migrations as well as a valid configuration, but it scales poorly. The constraint solving approach first finds a solution and then checks for a possible sequence of migrations.

Filieri et al. in [67] present a mathematical framework for run-time probabilistic model checking that, given a reliability model and a set of requirements, statically generates a set of expressions, which can be efficiently used at run-time to verify system requirements. This work focuses on computing efficiently

reliability properties in frequently changing and uncertain environments. Probabilistic model-checking plays a crucial role in evaluating reliability properties, typically expressed in PCTL, over a Discrete Time Markov Chain (DTMC) model of the running system.

In [68], Cámara et al. presents a formal verification approach at design-time based on model checking of stochastic multiplayer games (SMGs) in which adaptation latency is considered explicitly – *proactive latency-aware adaptation*. In [69], the approach is also brought to run-time and takes into account the inherent uncertainty of the environment predictions needed for looking ahead. Adaptation decisions in the model are left underspecifed through nondeterminism; a probabilistic model checker then resolves the non deterministic choices to deal with the infeasibility of adaptation options due to the latency or conflicts between them. The same mechanism allows the adaptation decision to select multiple adaptation strategies to execute in parallel when they do not interfere with each other.

The framework SCADE [70] allows modeling and verifying adaptive systems. The Lustre language, a typed synchronous dataflow language with a discrete time model, is used to specify functional requirements of the system. These are then verified by the SCADE Suite. Only safety and liveness properties can be checked directly.

In [71], a framework for modeling and analyzing distributed adaptive real-time systems using the process calculus Timed CSP (Communicating Sequential Processes) is presented. The framework allows to differentiate between functional data and adaptive control data. It also allows the modular verification of functional and adaptation behavior using the notion of process refinement in Timed CSP.

We divide approaches of the state of the art dealing with uncertainty in self-adaptive systems in two main groups: 1) those based on runtime automated verification, and 2) those based on runtime simulation. Representative works of the first category (runtime verification-based techniques) are [67] and [69] (already mentioned above) that allow reasoning stochastically about uncertainty. Essentially, they propose to equip the self-adaptive system with a stochastic system model that maintains up-to-date knowledge about the relevant qualities and uncertainties of the environment at run-time. Runtime verification of this stochastic model enables to calculate expected quality properties (e.g., likelihood of failures, expected response times) for different adaptation options, allowing the system to identify system configurations that comply with the required goals and to adapt itself accordingly. Simulation-based techniques for providing guarantees for self-adaptive systems at runtime have not been well studied yet [72]. A representative work of such a category is the modular approach presented in [73] for decision making in self-adaptive systems. It extends some of the concepts in [61, 62] to deal with run-time concerns and account also for probabilistic aspects in behavior to support on-the-fly changes of adaptation goals (changing goals at runtime is a challenging type of uncertainty). Essentially, it adopts distinct models for each relevant system quality combined with runtime simulation of the models and statistical techniques to select an adaptation option that satisfies the system goals. In summary, exhaustive verification suffers from the state space explosion problem. Simulation is less time and resource consuming than exhaustive verification. However, the tradeoff is that the guarantees are bounded to a certain level of accuracy [73]. In spite of these recent advances, managing uncertainty in real-time self-adaptive systems is still an impervious engineering problem.

7.2. A comparison of frameworks for timed SA

Our work has been influenced mostly by the work of J. Zhang and B. Cheng [13]. Our framework extends the adaptation models originally presented in [12] and then instantiated using Petri Nets in [13] to deal with concerns associated with time. In particular, we include temporal constraints in the modeling phase and different semantics associated with events in order to model both mandatory actions (*strong* events, e.g., recovery after undesired behavior) and actions that may occur but are not forced to (*weak* events, e.g., undesired behavior). Therefore, starting from [13], we instantiated our timed adaptation models with TB nets, that represent a very expressive formalism for describing real-time or even time-critical systems. We then introduced a particular analysis technique able to construct the overall reachability graph with temporal information upon events, partitioned into disjoint regions representing different steady-state behaviors of a self-adaptive real-time system. By means of cross-zone transition properties, we allow therefore the verification of properties of interest with respect to SA that, as advocated in [8], typically map to transitions between different zones.

To compare our approach and the other frameworks described in Subsection 7.1, we adopt the following main and generally agreed features[9] of a formalism for specifying and verifying self-adaptive systems with time constraints:

- Basis: The underlying adopted formalism.
- Time model: discrete or continuous.
- Support for feedback control loops: feedback loops for adaptation (such as the MAPE-K control loop [3]) are considered cornerstones for the design of self-adaptive systems [4, 60, 9], since it is generally agreed that feedback loops facilitate the identification of the core phenomena to control and the realization of compositional verification techniques that can be applied incrementally along the adaptation loop [60]. Therefore, a formal framework should allow their modeling and analysis explicitly.
- Support at design-time and/or at run-time: formal methods are applied during system design or maintenance activities (offline), or even used by the system itself at run-time.
- Types of properties for formal verification: structural properties (that can be determined from the structure of the model without executing it); behavioral properties such as safety, liveness, reachability, deadlock, and some of their variations specific to adaptation (i.e. desired characteristics of the adaptation logic, such as robustness for self-healing systems).
- Support for uncertainty: the ability of approaches for the specification and analysis of self-adaptive software systems to explicitly incorporate the uncertainty underlying the adaptation.

Framework	Basis	Time model	Feedback loops	Design/ run-time	Structural prop.	Behavioral prop.	Uncertainty
J. Zhang and B. Cheng [13]	P/T nets, LTL	✗	✗	design-time	✗	✓	✗
D. G. de la Iglesia and D. Weyns [61]	Timed-automata, TCTL	continuous	✓	design-time	✗	✓	✗
ActivFORMS [62]	Timed-automata, TCTL	discrete	✓	run-time	✗	✓	✗
Mechatronic UML [64]	UML	continuous	✗	design-time	✗	✓[a]	✗
SA-CIRCA approach [65]	non-deterministic finite automaton	continuous (by the real-time scheduler)	✗	run-time	✗	✓[b]	✗
M. Zeller and C. Prehofer [66]	state transition systems	continuous	✗	run-time	✗	✓[c]	✗
Run-time probabilistic model checking [67]	DTMCs, PCTL	discrete	✗	both	✗	✓[d]	✓
SCADE [70]	Lustre	discrete	✗	design-time	✗	✓[e]	✗
Timed CSP-based Framework [71]	Timed CSP	continuous	✗	design-time	✗	✓	✗
Proactive latency-aware adaptation [69]	Markov decision processes (MDPs)	discrete	✓	run-time	✗	✓	✓
Model-based simulation at runtime [73]	stochastic timed automata (STA)	continuous	✓	run-time	✗	✗	✓
ZAFETY framework (our approach)	TB nets, TCTL	continuous	✗	design-time	✓	✓	✓[f]

[a] In the general form of hybrid systems considered here reachability is undecidable. [b] Safety and Reachability.
[c] Domain-specific adaptation properties. [d] No adaptation properties. [e] Safety and Liveness.
[f] Limited to external uncertainty.

Table 4: A comparison of formal frameworks for timed SA.

[9] We did not attempt an exhaustive list of features for reviewing existing approaches, but only those features we consider relevant for our research goal.

Table 4 below summarizes the results of such a comparison by reporting the main features of the frameworks we considered in our work (see Subsection 7.1), including our proposal (the ZAFETY framework).

As main observation, we cannot identify a clear trend in the use of different formal modeling languages (e.g., transition systems, automata, state machines, Petri nets, Markov models, graphs, and process algebras) because they are roughly equally diffuse.

Some approaches adopt a discrete time model, others adopt a continuous time model. Clearly, this latter choice is more appropriate for specifying and verifying real-time systems.

Most of the formalization approaches mentioned above (including our framework) do not support the explicit modeling of feedback loops for SA and their properties. The actual feedback control loops are hidden or abstracted.

Some approaches use formal methods at design-time (offline), others at run-time or in combination. We also observed that a number of approaches exploit tools conceived for offline analysis to perform run-time analysis. This observation suggests that there is a lack of light-weight pluggable tools to support formal verification at run-time of SA with time constraints. Our primary research focus was grounded in providing assurances for system requirements at design time, but we want to extend the approach to deal also with run-time concerns.

The use of formal methods for providing correctness of system structural properties (like in our approach) is usually not considered. Most of the verification approaches focus on checking the classical range of behavioral properties, while the number of approaches that consider behavioral properties specific to the adaptation concerns remains limited.

Managing uncertainty and also account for timing aspects is very challenging. Most of the promising approaches dealing with uncertainty in self-adaptive systems are based on stochastic behaviors (such as works [67] and [69]) for making decisions under probability theory. Probabilistic, nondeterministic and real-time characteristics seem to be essential for modeling self-adaptive systems; however probabilistic models are known to be computationally expensive for execution, which makes them unsuitable for use at runtime, where often decisions have to be made very fast [74]. Runtime simulation-based techniques, such as the approach in [73], have not been well studied yet [72].

In general, the use of timed formalisms for modeling and verifying real-time requirements of self-adaptive systems remains limited. This indicates that current research on formal methods in real-time self-adaptive systems requires further investigation.

8. Threats to validity

Regarding threats to the validity of our work, we have identified and tried to address the following issues. First, there is the risk that the algorithms and the tool mistakenly prove the system requirements expressed through both structural and behavioral properties. Thus, we applied intensive unit testing and integration testing activities to mitigate this risk. Moreover, the ZAFETY software tool has been developed starting from a consolidated framework (i.e., the GRAPHGEN software tool [32, 75]).

Regarding the validity of the generalization of our approach, having defined our adaptation models in terms of state space (TRG for TB nets) reduces the risk that our formalization holds only thanks of the used notation. There is the risk that the proposed zone-based methodology can be applied only to few examples (or to a very limited form of adaptation). We have addressed this risk by applying our method to different case studies including two very common and representative real-time system examples (i.e., the GB system [15] and the UAV system, introduced in this article). Moreover, we compared our work with many other successful state-of-the-art approaches, borrowing some consolidated concepts from them. We believe that the PN-based formal specification technique is general enough to be successfully used to design (distributed) real-time self-adaptive systems, with the desired level of abstraction. Moreover, the proposed adaptation models are well accepted in the literature [12, 13], so the risk that they are biased by our running examples is reduced. In general, we believe our zone-based setting for modeling adaptation can be easily integrated with any transition-based formalism (e.g., different PN extensions or different automata-based formalisms).

9. Conclusion and future directions

This article presents a formal framework for specifying and verifying the behavior of real-time self-adaptive systems. The framework introduces the Zone-based TB Petri nets formalism that combines a *zone-based* specification approach with timed adaptation models. These last have been defined by extending the adaptation models presented in [13] with temporal constraints and different temporal semantics for modeling both mandatory and optional timed events.

Our zone-based modeling approach allows functional aspects to be kept separated from adaptation aspects. Zones, describing different steady-state behaviors of the system, can then be used either in isolation to verify non-adaptive behavior by means of *intra-zone properties*, or all together, to verify global system requirements through *inter-zone properties*. The verification of timing requirements is supported through *timed properties*, able to check that both functional aspects and adaptation comply with specific temporal deadlines. In addition, the framework supports interesting (timed) *robustness properties*, to ensure self-healing capability that represents a very important issue when dealing with real-time or even time-critical systems.

The proposed framework has been implemented as a Java software tool, called ZAFETY. We have shown the effectiveness of our framework by modeling and verifying the self-healing behavior of some time-critical systems. Here, we presented a significant case study: the UAV system, used as running example to illustrate how our formal framework can prove correctness of functional, adaptation, self-healing, and temporal aspects.

The research presented in this article focuses on providing functional requirements assurance for real-time self-adaptive systems at design-time. Developing new run-time verification techniques or revising existing verification techniques that work mainly at design-time (e.g., model checking) is still challenging [76]. We are currently investigating on techniques enabling run-time verification of real-time programs by checking conformance of timed events through a simultaneous execution of the corresponding TB net model. Our goal is to exploit this run-time verification technique to enable the use of the proposed formal framework also at run-time for a continuous verification of the run-time adaptation within predetermined time constraints. Preliminary results are promising. In particular, work presented in [27] introduces MahaRAJA, i.e., an event-based runtime verification framework that makes use of TB Petri nets. Experiments show that MahaRAJA is highly scalable and it introduces bounded overhead and effectively reduces the involvement of the monitor at run time by using negligible auxiliary memory [77].

As future work, we want also to extend our zone-based modeling approach in order to incorporate feedback loops (such as the MAPE-K control loop model [3, 60]) explicitly in our TB net models. Modeling feedback loops as zones would facilitate the identification of the core phenomena to control as they provide real separation of concerns between the domain and the adaptation concerns. To this purpose, we want to take inspiration from the recent results presented in [78] aiming at formally specifying decentralized adaptation control of MAPE-K feedback loops in terms of multi-agent *Abstract State Machines* [79]. Feedback loops would also allow the realization of compositional verification techniques, including timing constraints, that could be applied incrementally along the adaptation loop structure [60]. It would be therefore possible to verify properties of the feedback loops under time constraints. To this last purpose, we want to combine the proposed zone-based TB net formalism with the run-time verification framework already presented in [27].

Finally, in the future we would like also to extend our framework to manage a more comprehensive form of uncertainty in real-time self-adaptive systems. To this purpose, we want to combine our zone-based modeling technique with purely stochastic formalisms to express and verify *soft* real-time constraints in presence of uncertainty. A modeling formalism of choice in this field is *Generalized Stochastic Petri Nets* (GSPNs) [80]. In this case, we map regions and adaptation sets into a Continuous Time Markov chain [81] (i.e., the reachability graph of a GSPN model). Alternatively, we could exploit approaches that combine techniques from both fields of real-time verification and probabilistic verification. In particular we could exploit timed extensions of Petri nets with stochastic evolution rules, such as *Stochastic Timed Petri Nets* (STPNs) [82]. This would allow the specification and verification of both *hard* timing constraints (i.e., temporal deadlines) and randomized delays.

References

[1] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al., Software engineering for self-adaptive systems: A research roadmap, Springer, 2009.

[2] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al., Software engineering for self-adaptive systems: A second research roadmap, in: Software Engineering for Self-Adaptive Systems II, Springer, 2013, pp. 1–32.

[3] J. O. Kephart, D. M. Chess, The vision of autonomic computing, IEEE Computer 36 (1) (2003) 41–50.

[4] M. C. Huebscher, J. A. McCann, A survey of autonomic computing – degrees, models, and applications, ACM Comput. Surv. 40 (3) (2008) 7:1–7:28.

[5] N. Esfahani, S. Malek, Uncertainty in self-adaptive software systems, in: R. de Lemos, H. Giese, H. A. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers, Vol. 7475 of Lecture Notes in Computer Science, Springer, 2010, pp. 214–238.

[6] D. Perez-Palacin, R. Mirandola, Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation, in: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14, ACM, New York, NY, USA, 2014, pp. 3–14.

[7] M. Camilli, A. Gargantini, P. Scandurra, C. Bellettini, Towards Inverse Uncertainty Quantification in Software Development (Short Paper), Springer International Publishing, Cham, 2017, pp. 375–381. doi:10.1007/978-3-319-66197-1_24.

[8] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, T. Ahmad, A survey of formal methods in self-adaptive systems, in: B. C. Desai, E. Vassev, S. P. Mudur, B. C. Desai (Eds.), C3S2E, ACM, 2012, pp. 67–79.

[9] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511), Dagstuhl Reports 3 (12) (2014) 67–96.

[10] G. D. Rodosek, K. Geihs, H. Schmeck, B. Stiller, Self-Healing Systems: Foundations and Challenges, no. 09201 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, 2009.

[11] C. Ghezzi, D. Mandrioli, S. Morasca, M. Pezzè, A unified high-level Petri net formalism for time-critical systems, IEEE Trans. Softw. Eng. 17 (1991) 160–172.

[12] J. Zhang, B. H. C. Cheng, Specifying adaptation semantics, SIGSOFT Softw. Eng. Notes 30 (4) (2005) 1–7.

[13] J. Zhang, B. H. C. Cheng, Model-based development of dynamically adaptive software, in: Proc. of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 371–380.

[14] G. Gardey, O. H. Roux, O. F. Roux, State space computation and analysis of time petri nets, Theory Pract. Log. Program. 6 (3) (2006) 301–320.

[15] M. Camilli, A. Gargantini, P. Scandurra, Specifying and verifying real-time self-adaptive systems, in: 26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015, IEEE, 2015, pp. 303–313.

[16] J. L. Peterson, Petri nets, ACM Computing Surveys 9 (3) (1977) 223–252.

[17] B. Berthomieu, M. Diaz, Modeling and verification of time dependent systems using time Petri nets, IEEE Trans. Softw. Eng. 17 (1991) 259–273.

[18] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, in: Lectures on Concurrency and Petri Nets: Advances in Petri Nets, Springer Berlin Heidelberg, 2004, pp. 87–124.

[19] Y. Gurevich, Sequential abstract-state machines capture sequential algorithms, ACM Trans. Comput. Logic 1 (1) (2000) 77–111.

[20] C. Ramchandani, Analysis of asynchronous concurrent systems by timed Petri nets, Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1974).

[21] W. J. Lee, S. D. Cha, Y. R. Kwon, Integration and analysis of use cases using modular petri nets in requirements engineering, Software Engineering, IEEE Transactions on 24 (12) (1998) 1115–1130.

[22] D. G. D. L. Iglesia, D. Weyns, Mape-k formal templates to rigorously design behaviors for self-adaptive systems, ACM Trans. Auton. Adapt. Syst. 10 (3) (2015) 15:1–15:31.

[23] M. Camilli, C. Bellettini, L. Capra, M. Monga, CTL model checking in the cloud using mapreduce, in: Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2014, IEEE CS Press, Los Alamitos, CA, USA, 2014, to appear.

[24] M. Camilli, Formal verification problems in a big data world: Towards a mighty synergy, in: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, ACM, New York, NY, USA, 2014, pp. 638–641. doi:10.1145/2591062.2591088.
URL http://doi.acm.org/10.1145/2591062.2591088

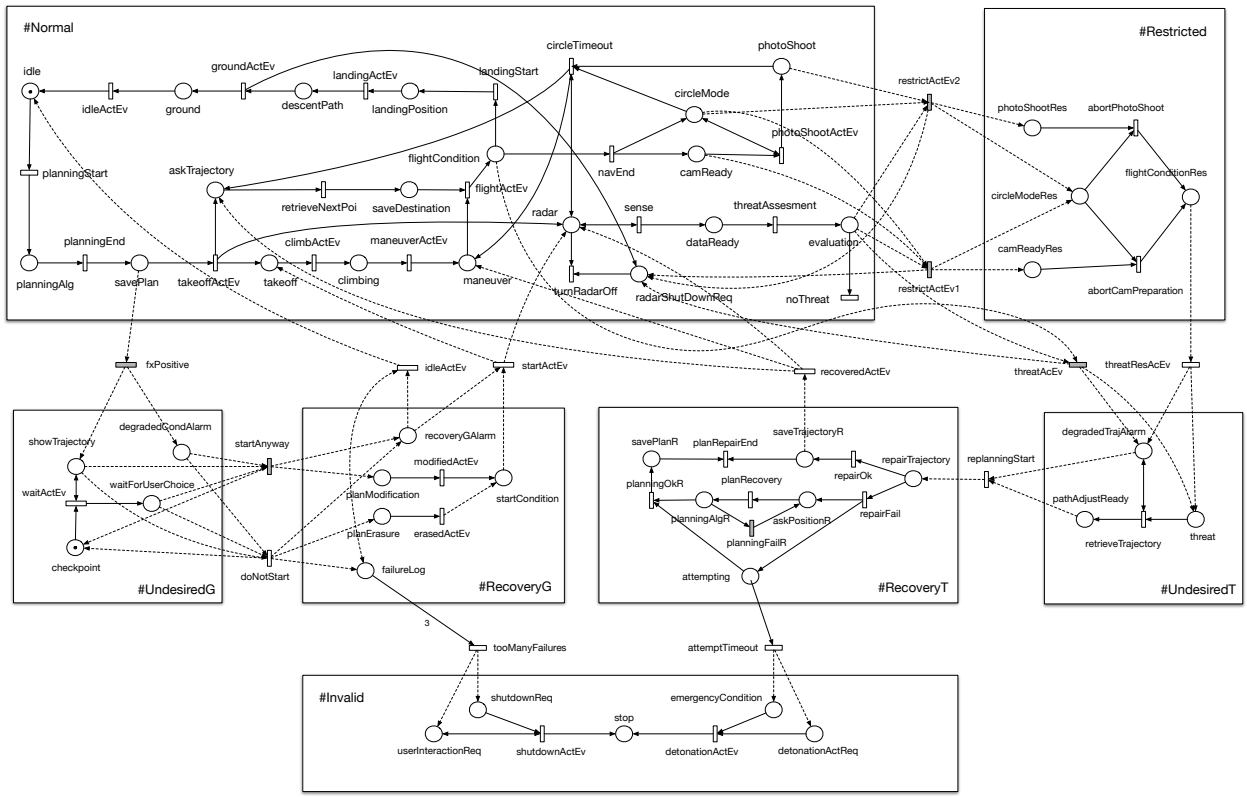
[25] C. Bellettini, M. Camilli, L. Capra, M. Monga, Distributed ctl model checking using mapreduce: Theory and practice, Concurr. Comput. : Pract. Exper. 28 (11) (2016) 3025–3041. doi:10.1002/cpe.3652.
URL https://doi.org/10.1002/cpe.3652

[26] M. Camilli, C. Bellettini, L. Capra, M. Monga, A formal framework for specifying and verifying microservices based process flows, in: A. Cerone, M. Roveri (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2018, pp. 187–202.

[27] M. Camilli, A. Gargantini, P. Scandurra, C. Bellettini, Event-based runtime verification of temporal properties using time basic petri nets, in: C. Barrett, M. Davies, T. Kahsai (Eds.), NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings, Vol. 10227 of Lecture Notes in Computer Science, Springer International Publishing, 2017, pp. 115–130.

[28] M. Camilli, C. Bellettini, L. Capra, Design-time to run-time verification of microservices based applications, in: A. Cerone, M. Roveri (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2018, pp. 168–173.
[29] M. Camilli, C. Bellettini, L. Capra, M. Monga, Coverability analysis of time basic petri nets with non-urgent behavior, in: J. H. Davenport, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, D. Zaharie (Eds.), 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, September 24-27, 2016, IEEE Computer Society, 2016, pp. 165–172.
[30] L. Pan, Z. J. Ding, M. C. Zhou, A configurable state class method for temporal analysis of time petri nets, IEEE Transactions on Systems, Man, and Cybernetics: Systems 44 (4) (2014) 482–493. doi:10.1109/TSMC.2013.2258907.
[31] M. Boyer, O. H. Roux, Comparison of the Expressiveness of Arc, Place and Transition Time Petri Nets, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 63–82. doi:10.1007/978-3-540-73094-1_7.
[32] C. Bellettini, L. Capra, Reachability analysis of time basic Petri nets: A time coverage approach, 2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing 0 (2011) 110–117.
[33] C. Bellettini, M. Camilli, L. Capra, M. Monga, Symbolic state space exploration of RT systems in the cloud, in: Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, IEEE CS Press, Los Alamitos, CA, USA, 2012, pp. 295–302.
[34] R. W. Floyd, Algorithm 97: Shortest path, Commun. ACM 5 (6) (1962) 345–.
[35] J. Horen, Linear programming, by katta g. murty, john wiley & sons, new york, 1983, 482 pp, Networks 15 (2) (1985) 273–274.
[36] H. Boucheneb, R. Hadjidj, CTL* model checking for time Petri nets, Theor. Comput. Sci. 353 (1) (2006) 208–227.
[37] C. Bellettini, M. Camilli, L. Capra, M. Monga, Mardigras: Simplified building of reachability graphs on large clusters, in: P. Abdulla, I. Potapov (Eds.), Reachability Problems, Vol. 8169 of LNCS, Springer Berlin Heidelberg, 2013, pp. 83–95.
[38] R. Allen, R. Douence, D. Garlan, Specifying and analyzing dynamic software architectures, in: E. Astesiano (Ed.), Fundamental Approaches to Software Engineering, Vol. 1382 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1998, pp. 21–37.
[39] G. Vachtsevanos, B. Ludington, Unmanned aerial vehicles: Challenges and technologies for improved autonomy, in: Proceedings of the 10th WSEAS International Conference on Systems, ICS'06, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2006, pp. 56–63.
[40] P. Schaefer, R. Colgren, R. Abbott, H. Park, A. Fijany, F. Fisher, M. James, S. Chien, R. Mackey, M. Zak, T. Johnson, S. Bush, Reliable autonomous control technologies (react) for uninhabited air vehicles, in: Aerospace Conference, 2001, IEEE Proceedings., Vol. 2, 2001, pp. 2/677–2/684 vol.2.
[41] P. Gunetti, T. Dodd, H. Thompson, Simulation of a soar-based autonomous mission management system for unmanned aircraft, Journal of Aerospace Information Systems 10 (2) (2013) 53–70.
[42] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (2009) 14:1–14:42.
[43] R. de Lemos, J. L. Fiadeiro, An architectural support for self-adaptive software for treating faults, in: Proceedings of the First Workshop on Self-healing Systems, WOSS '02, ACM, New York, NY, USA, 2002, pp. 39–42.
[44] P. Robertson, R. Laddaga, Model based diagnosis and contexts in self adaptive software, in: O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, M. van Steen (Eds.), Self-star Properties in Complex Information Systems, Vol. 3460 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 112–127.
[45] R. Tarjan, Depth-first search and linear graph algorithms, in: Switching and Automata Theory, 1971., 12th Annual Symposium on, 1971, pp. 114–121.
[46] E. Clarke, E. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: D. Kozen (Ed.), Logics of Programs, Vol. 131 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1982, pp. 52–71.
[47] R. Alur, C. Courcoubetis, D. Dill, Model-checking in dense real-time, Inf. Comput. 104 (1) (1993) 2–34.
[48] P. Bouyer, Model-checking timed temporal logics, Electronic Notes in Theoretical Computer Science 231 (2009) 323 – 341.
[49] H. Boucheneb, G. Gardey, O. H. Roux, Tctl model checking of time petri nets, J. Logics and Computing 19 (6) (2009) 1509–1540.
[50] N. J. Dingle, W. J. Knottenbelt, T. Suto, Pipe2: a tool for the performance evaluation of generalised stochastic Petri nets, SIGMETRICS Perform. Eval. Rev. 36 (4) (2009) 34–39.
[51] L. M. Hillah, F. Kordon, L. Petrucci, N. Trèves, Pnml framework: An extendable reference implementation of the petri net markup language, in: Proceedings of the 31st International Conference on Applications and Theory of Petri Nets, PETRI NETS'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 318–327.
[52] E. R. Gansner, S. C. North, An open graph visualization system and its applications to software engineering, SOFTWARE - PRACTICE AND EXPERIENCE 30 (11) (2000) 1203–1233.
[53] R. Allen, R. Douence, D. Garlan, Specifying and analyzing dynamic software architectures, in: FASE, 1998, pp. 21–37.
[54] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.
[55] B. Morin, O. Barais, G. Nain, J. Jézéquel, Taming dynamically adaptive systems using models and aspects, in: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE, 2009, pp. 122–132. doi:10.1109/ICSE.2009.5070514.
URL http://dx.doi.org/10.1109/ICSE.2009.5070514
[56] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, R. Mirandola, MOSES: A framework for qos driven runtime adaptation of service-oriented systems, IEEE Trans. Software Eng. 38 (5) (2012) 1138–1159.
[57] R. Mirandola, P. Potena, P. Scandurra, Adaptation space exploration for service-oriented applications, Science of Computer Programming 80, Part B (0) (2014) 356–384.
[58] Z. Ding, Y. Zhou, M. Zhou, Modeling self-adaptive software systems with learning petri nets, IEEE Trans. Systems, Man,

and Cybernetics: Systems 46 (4) (2016) 483–498.
[59] M. U. Iftikhar, D. Weyns, A case study on formal verification of self-adaptive behaviors in a decentralized system, in: Proc. 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012., Vol. 91 of EPTCS, 2012, pp. 45–62.
[60] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, M. Shaw, Software engineering for self-adaptive systems, in: B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems, Springer-Verlag, Berlin, Heidelberg, 2009, Ch. Engineering Self-Adaptive Systems Through Feedback Loops, pp. 48–70.
[61] D. G. de la Iglesia, D. Weyns, Guaranteeing robustness in a mobile learning application using formally verified MAPE loops, in: M. Litoiu, J. Mylopoulos (Eds.), Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2013, San Francisco, CA, USA, May 20-21, 2013, IEEE Computer Society, 2013, pp. 83–92.
[62] M. U. Iftikhar, D. Weyns, Activforms: Active formal models for self-adaptation, in: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, ACM, New York, NY, USA, 2014, pp. 125–134.
[63] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Lectures on Concurrency and Petri Nets, Vol. 3098 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 87–124.
[64] H. Giese, W. Schäfer, Assurances for Self-Adaptive Systems: Principles, Models, and Techniques, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, Ch. Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML, pp. 152–186.
[65] D. J. Musliner, Imposing real-time constraints on self-adaptive controller synthesis, in: in Proc. Int'l Workshop on Self-Adaptive Software, Springer-Verlag, 2000, pp. 143–160.
[66] M. Zeller, C. Prehofer, Timing constraints for runtime adaptation in real-time, networked embedded systems, in: H. A. Müller, L. Baresi (Eds.), 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012, Zurich, Switzerland, June 4-5, 2012, IEEE, 2012, pp. 73–82.
[67] A. Filieri, C. Ghezzi, G. Tamburrelli, Run-time efficient probabilistic model checking, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 341–350.
[68] J. Cámara, G. A. Moreno, D. Garlan, Stochastic game analysis and latency awareness for proactive self-adaptation, in: G. Engels, N. Bencomo (Eds.), 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings, Hyderabad, India, June 2-3, 2014, ACM, 2014, pp. 155–164.
[69] G. A. Moreno, J. Cámara, D. Garlan, B. R. Schmerl, Proactive self-adaptation under uncertainty: a probabilistic model checking approach, in: E. D. Nitto, M. Harman, P. Heymans (Eds.), Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015, ACM, 2015, pp. 1–12.
[70] M. Gudemann, A. Angerer, F. Ortmeier, W. Reif, Modeling of self-adaptive systems with scade, in: Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on, 2007, pp. 2922–2925.
[71] T. Göthel, V. Klös, B. Bartels, Modular design and verification of distributed adaptive real-time systems based on refinements and abstractions, EAI Endorsed Trans. Self-Adaptive Systems 1 (1) (2015) e5.
[72] D. Weyns, N. Bencomo, R. Calinescu, J. Cámara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, G. Tambrrellii, Perpetual assurances for self-adaptive systems, in: R. de Lemos, D. Garlan, C. Ghezzi, H. Giese (Eds.), Software Engineering for Self-Adaptive Systems (SEfSAS) 3, no. 9640 in Lecture Notes in Computer Science, Springer, 2017, to appear.
[73] D. Weyns, M. U. Iftikhar, Model-based simulation at runtime for self-adaptive systems, in: 2016 IEEE International Conference on Autonomic Computing (ICAC), IEEE, 2016, pp. 364–373.
[74] N. Esfahani, S. Malek, Uncertainty in self-adaptive software systems, in: R. de Lemos, H. Giese, H. A. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 214–238.
[75] M. Camilli, Petri nets state space analysis in the cloud, in: Software Engineering (ICSE), 2012 34th International Conference on, 2012, pp. 1638–1640.
[76] R. Calinescu, S. Kikuchi, Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems: 16th Monterey Workshop 2010, Redmond, WA, USA, March 31- April 2, 2010, Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, Ch. Formal Methods @ Runtime, pp. 122–135.
[77] The MahaRAJA framework, `https://maharajaframework.bitbucket.io/`, accessed: Dec. 2017.
[78] P. Arcaini, E. Riccobene, P. Scandurra, Formal design and verification of self-adaptive systems with decentralized control, ACM Transactions on Autonomous and Adaptive Systems 11 (4) (2017) 25:1–25:35.
[79] E. Borger, R. F. Stark, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
[80] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, Modelling with Generalized Stochastic Petri Nets, 1st Edition, John Wiley & Sons, Inc., New York, NY, USA, 1994.
[81] J. Norris, Markov Chains, Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, 1998.
[82] G. Juanole, L. Gallon, Formal modelling and analysis of a critical time communication protocol, in: Factory Communication Systems, 1995. WFCS '95, Proceedings., 1995 IEEE International Workshop on, 1995, pp. 107–115.



Initial marking: $idle\{T_0\}$, $checkpoint\{T_0\}$
Initial constraint: $0 \leq T_0 \leq 10$

Figure A.18: The zone-based TB net model of the UAV case study. Weak transitions are depicted in gray.

Appendix A. The Complete UAV zone-based formal specification

This appendix contains the complete formal specification of the UAV system, used as running example throughout the article. In particular, Figure A.18 shows the complete zone-based TB net model for the management of an observation mission. Our zone-based approach is used to model the vehicle behavior. During the nominal behavior (i.e., the zone `#Normal`), the flight path can intersect any unknown entity that represents a threat such as hostile presences but also bad weather areas or no-fly zones. Therefore the UAV adapts itself to work in different degraded conditions, i.e., it eventually changes the flight plan in order to take a detour around the threats. Path adjustment during the mission planning (while the UAV is still on ground), are modeled by the zones `#UndesiredG` and `#RecoveryG`, while the path adjustment during the travel are modeled by the zones `#Restricted`, `#UndesiredT`, and `#RecoveryT`. The zone `#Invalid` models loss of functionality from which the recovery is no longer possible.