Specifying and Verifying Real-Time Self-Adaptive Systems

Matteo Camilli, Angelo Gargantini, Patrizia Scandurra
Department of Management, Information and Production Engineering (DIGIP)
Università degli Studi di Bergamo, Italy
Email: {matteo.camilli,angelo.gargantini,patrizia.scandurra}@unibg.it

***Abstract*—Self-adaptive systems autonomously adapt their behavior at run-time to react to internal dynamics and to uncertain and changing environment conditions. Specification and verification of self-adaptive systems are generally very difficult to carry out due to their high complexity, especially when involving time constraints. In the last case, in fact, the correctness of systems depends also on the time associated with events.**

This paper introduces a formal approach to specify and verify the self-adaptive behavior of real-time systems. Our specification formalism is based on Time-Basic Petri nets, a particular timed extension of Petri nets. We propose *adaptation models* to realize self-adaptation with temporal constraints and we adopt a *zone-based modeling* approach to support separation of concerns during the modeling phase. Zones identified during the modeling phase can be then used as modules (TB Petri subnets) either in isolation, to verify *intra-zone properties*, or all together, to verify *inter-zone properties* over the entire system model and check that all the temporal deadlines are met. We illustrate our approach by modeling and verifying a time-critical Gas Burner system that exhibits a self-healing behavior.

I. Introduction

Modern advanced 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 such changes autonomously [10], [12], [18]. They aim at achieving particular quality goals and ensuring the required functionality in a *fail-soft* manner even in hostile or error conditions realizing the so called self-* properties (self-optimization, self-healing, self-protection, etc.).

The development of such *self-adaptive systems* is extremely challenging and demands new formal approaches that can efficiently tackle the problems of expressing autonomy requirements and ensuring the functional correctness of the system's adaptation logic both at design time and at runtime. However, the survey in [26] shows that, although the attention for self-adaptive software systems is gradually increasing, the number of studies that employ formal methods remains low, and mainly related to runtime verification. Specification and verification of self-adaptive systems are very difficult to carry out, especially when involving time constraints. In fact, in the latter case, the functional correctness of the system and of its adaptation logic depends 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.

This paper introduces a formal approach to specify and verify the behavior of real-time self-adaptive systems. Our specification formalism is based on the *Time-Basic (TB) Petri nets* (or simply TB net) [15], a particular timed extension of Petri nets. We provide some enhancements to the TB net formalism to deal with self-adaptive systems and real-time constraints. We adopt and extend the *adaptation models – one-point adaptation*, *overlapp adaptation*, and *guided adaptation* – originally presented in [27], [28] to realize self-adaptation with temporal constraints in TB Petri nets. Moreover, the proposed specification approach allows for separation of concerns during the modeling phase by dividing the system's TB Petri net model into *zones*[1] *– zone-based TB Petri nets*. Our verification approach allows for checking timed events through the symbolic execution of the system's TB petri net, thus it allows also the verification of timed adaptation. Zones of the TB Petri net identified during the modeling phase can be used as modules (TB Petri subnets) either in isolation, to check *intra-zone properties*, or all together, to check *inter-zone properties* over the entire system model. We illustrate our approach by modeling and verifying a time-critical Gas Burner system that exhibits a self-healing behavior.

The rest of this paper is organized as follows. Sect. II provides some background on the TB Petri nets and introduces the Gas Burner system here taken as running case study. It also describes the adaptation models that we have adopted and extended for realizing self-adaptation in TB Petri nets. Sect. III presents the proposed zone-based TB Petri nets for self-adaptive systems. Sect. V 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 case study. Sect. VI presents related work. Sect. VII concludes the paper and outlines future directions of our work.

II. Background concepts

This section briefly introduces our running example, the TB net formalism, and existing adaptation models that inspired us to realize self-adaptation in TB nets.

[1]The term zone is here to be intended differently from the forward zone-based reachability analysis of Time Petri Nets [13], where a zone stands for a finite convex union of regions (i.e., a representation of clock values using equivalence classes).

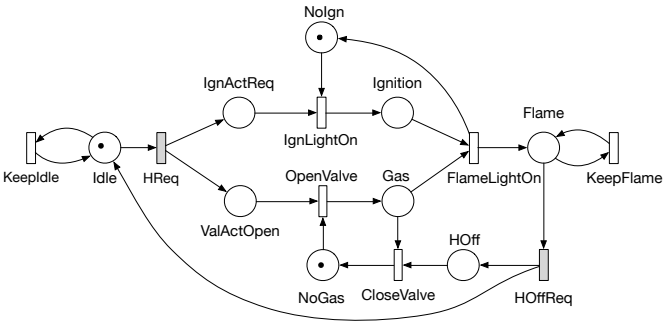
A. The Gas Burner Example

As running case study, we adopt the gas burner system since it represents a meaningful example of safety-critical system [21]. In a gas burner an accident may occur if an excessive amount of unburned gas leaks to the environment. In fact, a burning flame may be blown out causing some gas to leak before the failure is detected. The gas burner is controlled by a thermostat and the gas is ignited by an ignition transformer.

The normal behavior of the system follows these phases:

- **Idle**: Awaits heat request. No gas and ignition are supplied at this stage.
- **Ignite**: Ignition and gas supply start.
- **Burn**: Ignition is switched off, but gas is still supplied. The Burn phase is stable until heat request goes off. The Idle phase is then entered and the gas is turned off.

Moreover, a simple error recovery procedure is used. If a flame is not sensed within 1 sec. (ignite failure), or if the flame disappears during the Burn phase (flame failure), then the recovery phase is entered. Thus, the gas is turned off and an airing phase starts. If the system fails to recover within 5 sec., it raises an alarm and then stops its own execution.



Initial marking	Idle$\{T_0\}$ NoIgn$\{T_0\}$ NoGas$\{T_0\}$
Initial constraint	$0 \leq T_0 \leq 10$
KeepIdle	[Idle + 0.5 , Idle + 0.5]
HReq	[Idle + 0.1 , Idle + 0.5]
IgnLightOn	[IgnActReq + 0.1 , IgnActReq + 0.2]
OpenValve	[ValActOpen + 0.2, ValActOpen + 0.4]
FlameLightOn	[Gas + 0.02, max(Gas, Ignition) + 0.1]
KeepFlame	[Flame + 0.5, Flame + 0.5]
HOffReq	[Flame + 0.1, Flame + 0.5]
CloseValve	[HOff + 0.01, max(HOff, Gas) + 0.2]

Fig. 1. Simple Gas burner TB model. Weak transitions are depicted in gray.

B. Time Basic Petri Nets

TB nets belong to the category of Petri nets [22] in which system time constraints are expressed as numerical intervals 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 [6]), 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. Moreover, Petri Nets are known to be more scalable with respect to other formalisms for specifying highly concurrent systems [19].

The structure of a TB net is a triplet $\langle P, T, F \rangle$, where P and T are finite disjoint sets of *places* and *transitions*, respectively, and F is the flow relation, $F \subseteq (P \times T) \cup (T \times P)$. Given $v \in P \cup T$, let us denote ${}^\bullet v$ and $v^\bullet$, the backward and forward adjacent sets of v according to F, respectively, also called pre/post-sets of v.

We assume the domain of time-stamps to be $\mathbb{R}^+$. 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 $\mathbb{R}^+$ values. $f_t(en)$ represents the possible firing times of t. According to a *weak* semantics, t *can* fire at any instant $\tau \in f_t(en)$. A second interpretation states that t *must* fire at an instant $\tau \in f_t(en)$, unless it is disabled by the firing of any conflicting enabling tuple at an instant no greater than the latest firing time of t. Transitions with one such semantics are referred to as *strong*. Notice that the only possible semantics for Time Petri Nets [6] is strong. Hereafter, we denote a time function f_t with a pair of linear functions $[lb_t, ub_t]$, denoting parametric interval bounds.

As an example, consider the TB net in Figure 1. It represents the *normal* behavior (without failures) of the gas burner system. The initial marking represents the controller waiting for activation (*Idle* phase). If a heat request (represented by the firing of the weak transition *HReq*) occurs, the *Ignite* phase starts. The ignition actuator is represented by transitions *IgnLightOff* and *IgnLightOn*, and places *Ignition* and *NoIgn*. The flame is turned on if there are ignition and gas (transition *FlameLightOn*). When the flame goes on, the *burn* phase starts. At this point the system can either keep the flame on, or close the gas valve and return in idle phase, if a heat-off request is issued (represented by the firing of the weak transition *HOffReq*). The transition *FlameLightOn*, representing the system passing to burn state, can be interpreted as follows. It cannot fire before 0.02 time units elapse since the appearance of a token in place *Gas* (the minimum permanence time in ignite state). Moreover, the firing time cannot exceed the maximum between the time-stamp of the token in place *Gas* and the time-stamp of the token in place *Ignition* plus 0.1 (i.e., the system recognizes the presence of a flame within 0.1 time units).

GRAPHGEN [4], [2] is a powerful TB nets analysis software tool. The implemented technique aims at building a finite representation of the underling infinite state space. Where each reachable symbolic state $S = \langle M, C \rangle$ is composed of a *symbolic marking* M, representing symbolic time-stamps associated with tokens, and a *symbolic constraint* C, representing the relationships among symbolic time-stamps, through a set of linear inequalities. For instance, the initial symbolic state of the simple gas burner (Figure 1) is represented by

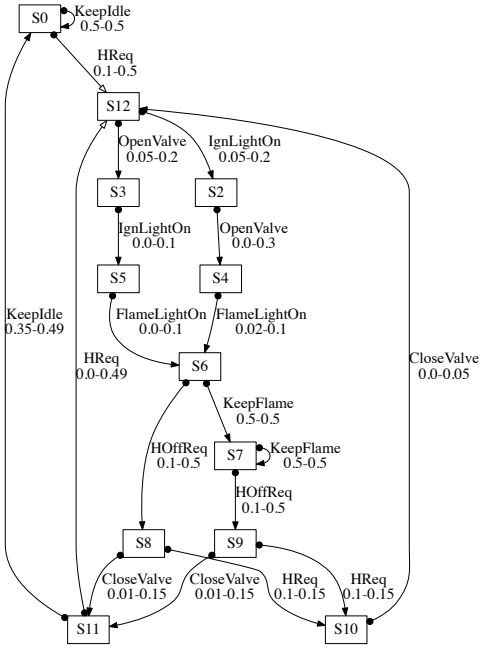


Fig. 2. Time reachability graph constructed from the model in Figure 1.

$S_0 = \langle M_0, C_0 \rangle$ such that:

$$\begin{aligned} M_0 &:= \text{Idle}\{T_0\}, \text{NoIgn}\{T_0\}, \text{NoGas}\{T_0\} \\ C_0 &:= T_0 \geq 0 \ \wedge \ T_0 \leq 10 \end{aligned}$$

thus, a concrete reachable state s is represented by $\mathcal{S}$, if and only if s is obtained from $\mathcal{S}$ by a numerical replacement of symbolic time-stamps being a solution of C.

Figure 2 shows the overall *time reachability graph* (TRG) computed from the model in Figure 1, using GRAPHGEN. Edges between symbolic states are labeled with the firing transition and two temporal values representing the local minimum and the local maximum firing time, respectively. Edges can be of different types representing the usual transition relationships among symbolic states [3]. A white arrow, for example the edge between S_0 and S_{12}, means that only a subset of the concrete states represented by S_{12} is reachable from S_0. 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, $E \subseteq N \times T \times N$ is the set of edges, and S_0 is the initial symbolic state.

C. Adaptation Models

The example shown in Figure 1 models the optimistic condition where ignition failure and flame failure do not happen. However, although both gas and ignition are supplied, the lighting may fail. Moreover, once the burning phase is stable, the flame may switch off due to external events (e.g., wind). Thus, the system should discover, diagnose, and react to these exceptions within strict time constraints in order to avoid great loss, such as damaging the surrounding physical environment or even threatening human lives. Therefore, we want to add a *self-healing subsystem* [11], [23] in order to formally specify a *self-adaptive* behavior [24] in charge of handling disruption to restore the system to normal conditions within hard deadlines.

Figure 2, shows the behavior over time of the gas burner example operating in its own *normal* domain (without faults). Anyway, a self-adaptive system operates in different domains and changes its behavior at runtime 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 exhibit a different *steady-state* behavior [1], [28]. 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* [27], [28].

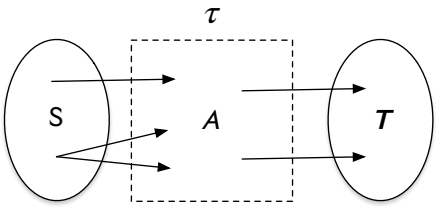


Fig. 3. A simple self-adaptive system with time constraints upon transitions.

a) One-point adaptation: The *one-point adaptation* process completes with a single edge e. Thus the steady-state behavior $\mathcal{S}$ should end within e and the steady-state behavior $\mathcal{T}$ should start immediately after e [27], [28]. The source states suitable for adaptation, with outgoing adaptive transitions, are called *quiescent states*.

b) Overlap adaptation: During the *overlap adaptation* process, the target behavior starts immediately after reaching $\mathcal{A}$, and the source behavior stops when we leave $\mathcal{A}$ and we enter the region $\mathcal{T}$. At this point the system exhibits only the target steady-state behavior [27], [28]. This means that within the boundaries of $\mathcal{A}$ the source and the target behaviors overlap. Anyway, the adaptive system should always satisfy the *adaptation integrity constraint*: once the adaptation process starts, it should eventually end and reach a state belonging to the target region.

c) Guided adaptation: The *guided adaptation* process starts with the fact that the system in its source steady-state can receive an adaptation request in non quiescent states. Therefore, it should enter a restricted mode, where some

functionality are blocked in order to reach a quiescent state and perform the transformation as fast as possible [27], [28]. In this case, the region $\mathcal{S}$ reaches $\mathcal{A}$ which represents the restricted mode. $\mathcal{A}$ eventually reaches a quiescent state that leads to $\mathcal{T}$ through a one-point adaptation.

In the next sections we extend these adaptation models and instantiate them with the TB nets formalism in order to deal with temporal constraints and different temporal semantics.

III. Zone-Based Modeling

To recover from flame failures, a *self-healing subsystem* must be added to the gas burner system modeled in Figure 1. The self-healing subsystem should be able to detect undesired behavior (both ignite and flame failures) and then adapt itself in order to restore the normal behavior. Therefore, the state space of the entire system should be characterized at least by four disjoint regions [16], representing the system exhibiting different steady-state behaviors. The four regions, sketched by Figure 4, are the *normal behavior*, where the system performs its main functionalities without faults; 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 (represented by dashed lines) carry out the transformations using different adaptation models.

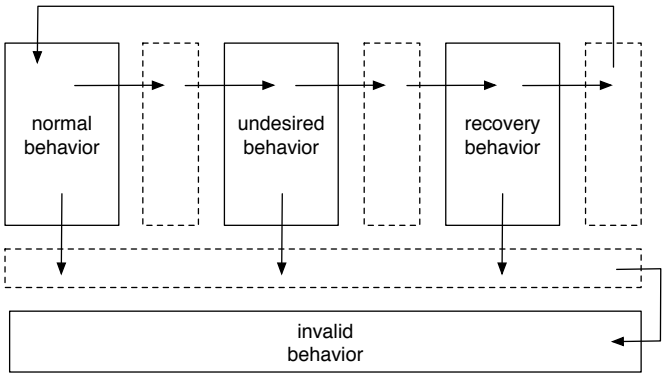


Fig. 4. State space regions denoting different behaviors of a self-adaptation.

A. Zone-Based TB Nets

In order to achieve these structural properties on the state space, we propose a *zone-based modeling* approach. This technique aims at identifying and isolating different modules of the system that abstract the adaptive behavior in order to reach a separation of concerns during both the modeling and the verification phases.

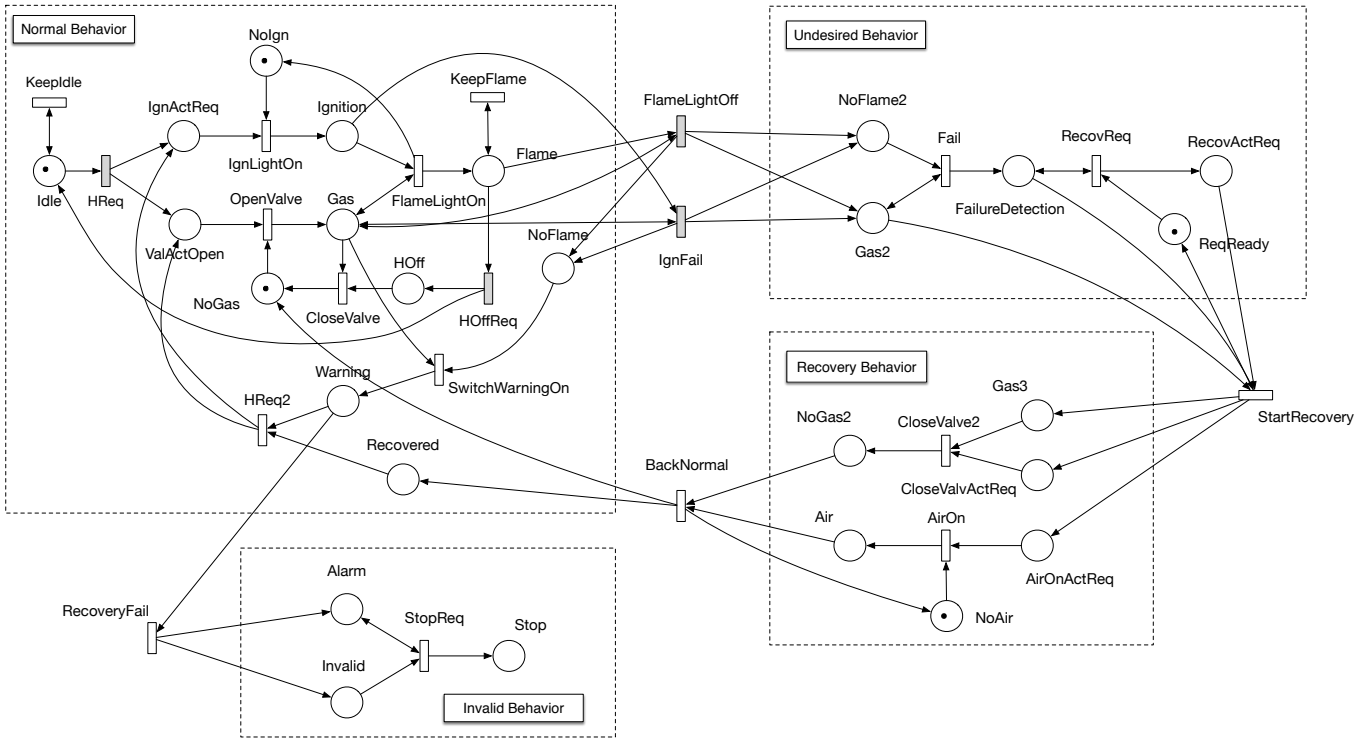
The entire zone-based model of the gas burner system is shown in Figure 5:

- *normal behavior*: This zone models the normal behavior of the system as described in section II-B. Few places and transitions has been added with respect to the model shown in Figure 1, in order to make the module able to interact with the rest of the system. The place *NoFlame* keeps track of failures and the place *Warning* becomes marked if *gas* leaks without *flame* are detected. The place *Warning* remains marked while the self-healing subsystem is running (the system exhibits undesired behavior and then adapt itself to recovery behavior). Whenever the recovery fails to restore the normal behavior within the *RecoveryFail* deadline ($[Warning + 2.0, Warning + 2.0]$ which corresponds to 2.0 time units after the enabling time of the transition *startRecovery*), the system enters the invalid behavior zone. It could be of interest to check some local invariants or local liveness properties to ensure correctness of this module in isolation. For instance, we can verify that the module never recognizes the presence of flame without gas (invariant); or that the burning phase is reachable within 1.8 time units from a heat request (timed bounded liveness property).
- *undesired behavior*: The undesired behavior starts after a failure. The module recognizes a failure within 0.1 time units after a flame or a ignition failure happen (*Fail* firing transition). After a failure detection a recovery request is issued between 0.05 and 0.1 time units (*RecoveryReq* firing transition). After that, the system can enter the recovery behavior zone. Considering the undesired behavior zone in isolation, it could be of interest, for instance, to verify that the recovery request cannot be issued without a gas leak (safety property); or that the system recognizes the presence of gas foreach reachable state (invariant).
- *recovery behavior*: The recovery behavior zone models the actions that the system have to perform to restore the normal conditions. The adaptation starts with the following marked places: *Gas3*, *CloseValvActReq*, *AirOnActReq* that represent the presence of gas leaks, a valve-close request and an air activation request, respectively. At this point the transitions *CloseValve* and *AirOn* must fire, leading the system in a purge phase which is in charge of decreasing the concentration of accumulated gas. After the time required by the purge phase the normal behavior is restored. Some interesting properties could be verified upon this zone in isolation, for instance: after issuing a valve-close request and an air-on request, the module always reaches the purge phase within 4.2 time units (time bounded liveness property).
- *invalid behavior*: The invalid zone models the behavior of the system that should never be assumed. In particular, whenever the self-healing subsystem fails to restore the normal behavior within the proper deadline, an alarm raises (*Alarm* place marked). After that, the system halts, avoiding new incoming heat requests to be processed.

A *Zone* can be formally characterized as follows:

Definition 1: **Subnet**. Given a TB net $\langle P, T, F \rangle$, the structure $\langle P_S, T_S, F_S \rangle$ is a *subnet* iff. $P_S \subseteq P$, $T_S \subseteq T$, $F_S \subseteq F$, $P_S \neq \emptyset$, $T_S \neq \emptyset$, and $F_S \neq \emptyset$.

Definition 2: **Subnet preset**. Given a subnet z, the preset ${}^\bullet z$ is the set of transitions that connect places outside z to places belonging to z.



Initial marking: $Idle\{T_0\},\ NoGas\{T_0\},\ NoIgn\{T_0\},\ ReqReady\{T_0\},\ NoAir\{T_0\}$
Initial constraint: $0 \leq T_0 \leq 10$

Time-Functions:

HReq	$[Idle + 0.1, Idle + 0.5]$	**KeepIdle**	$[Idle + 0.5, Idle + 0.5]$
OpenValve	$[ValActOpen + 0.1, ValActOpen + 0.4]$	**IgnLightOn**	$[IgnActReq + 0.1, IgnActReq. + 0.2]$
FlameLightOn	$[Gas + 0.02, max(Gas, Ignition) + 0.1]$	**CloseValve**	$[HOff + 0.01, max(HOff, Gas) + 0.15]$
KeepFlame	$[Flame + 0.5, Flame + 0.5]$	**HOffReq**	$[Flame + 0.1, Flame + 0.5]$
HReq2	$[enab + 0.05, enab + 1.0]$	**SwitchWarningOn**	$[enab + 0.01, enab + 0.1]$
FlameLightOff	$[Flame + 0.1, Flame + 0.5]$	**IgnFail**	$[Gas + 0.05, Gas + 0.1]$
RecovReq	$[FailureDetection + 0.05, FailureDetection + 0.2]$	**Fail**	$[enab + 0.05, enab + 0.12]$
StartRecovery	$[FailureDetection + 0.05, RecovActReq + 0.1]$	**CloseValve2**	$[enab + 0.2, enab + 0.4]$
AirOn	$[AirOnActReq + 0.2, AirOnActReq + 0.4]$	**BackNormal**	$[enab + 0.05, enab + 0.1]$
RecoveryFail	$[enab + 2.0, enab + 2.0]$	**StopReq**	$[enab + 0.02, enab + 0.05]$

Fig. 5. Zone-based model of the gas burner system. Weak transitions are depicted in gray.

Definition 3: **Subnet postset**. Given a subnet z, the postset $z^\bullet$ is the set of transitions that connect places of z to places outside z.

Definition 4: **Zone**. Given a TB net PN and a subnet $z = \langle P_S, T_S, F_S \rangle$, z is a *zone* iff. the TB net $\langle P_S, T', F' \rangle$ is *weakly connected*[2], where:

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

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

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, that allow the connection among different zones. As an example, consider the undesired behavior subnet in Figure 5 along with its own preset {*FlameLightOff*, *IgnFail*} (with outgoing egdes), and its own postset {*StartRecovery*} (with incoming edges). We obtain a weakly connected Petri net, therefore the undesired behavior is a valid zone.

Definition 5: **Cross-zone transition**. Given a TB net $\langle P, T, F \rangle$ and a transition $t \in T$, t is a *cross-zone transition* iff. there exists one and only one zone z, s.t. $t \in z^\bullet$ and there exists at least a zone z' s.t. $t \in {}^\bullet z'$.

Considering the example in Figure 5, the set of cross-zone transitions is {*FlameLightOff*, *IgnFail*, *StartRecovery*, *BackNormal*, *RecoveryFail*}.

Definition 6: **Zone-based TB net**. A *zone-based TB net* model PN_Z is a TB net $\langle P, T, F \rangle$, composed of non empty a set of zones Z, s.t.

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

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

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

Back to our running case study, we can individuate four disjoint zones: the *normal behavior* zone, the *undesired behavior*

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

zone, the *recovery behavior* zone, and the *invalid behavior* zone. There are no places outside these zones and transitions are either inside zones or cross-zone. Moreover, all zones have at least an non empty preset or postset, connecting them to the rest of the system. Therefore the gas burner model is a zone-based TB net.

Definition 7: **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, such that:

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

Intuitively, the labeling function λ, will be used to associate a firing transition representing an action or an event with a specific steady-state behavior represented by a zone.

B. Zone-based Timed Adaptation Models

Adaptation models allow dynamic changes of the system's behavior to be handled. In our context, we construct adaptation models by means of *cross-zone* transitions, along with their own temporal functions used to compute temporal constraints upon the dynamically adaptive behavior.

1) Timed One-point Adaptation: The *timed one-point adaptation* is modeled by the pair (t, f_t) composed of a cross-zone transition 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 models 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), anyway 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 this transition can be either weak or strong, for instance the transition between the normal and the undesired zone have weak semantics because the undesired behavior could happen but it is not forced to. Instead, the transition between the undesired zone and the adaptation zone, have strong semantics because we want to force the system to adapt itself, in order to tackle a undesired behavior.

As an example consider the transition *StartRecovery* in Figure 5. It models a timed one-point adaptation between the undesired zone and the adaptation zone. In fact, its firing consumes tokens in places *Gas2*, *FailureDetection*, *RecoveryActReq* (belonging to the source zone) and it produces tokens in places *Gas3*, *CloseValvActReq*, *AirOnActReq* (belonging to the target zone). Once fired, the recovery behavior is the only executable zone, although the place *ReqReady* is marked.

2) Timed Overlap Adaptation: The *timed overlap adaptation* is modeled by a cross-zone transition 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 $z_{\mathcal{T}}$ 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 [27].

As an example consider the transition *FlameLightOff* in Figure 5. It models a timed overlap adaptation between the normal zone and the undesired zone. Its firing produces tokens into some places belonging to the source zone (*Gas* and *NoFlame*), and some places belonging to the target zone (*Gas2* and *NoFlame2*). Once fired, the source and the target zone execute in parallel, in fact, both the *SwitchWarningOn* and the *Fail* transitions are enabled.

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 functionality 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 should define a set of (t, f_t) pairs $\mathcal{G}$, such that each t is a cross-zone transition with weak semantics. Each transition, along with the temporal function, should connect places belonging to the original zone with places belonging to the restricted zone. This way we can 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 more is the size of $\mathcal{G}$, the more is the responsiveness of the system because we increase the number of states able to handle adaptation requests.

For the sake of readability, the example shown in Figure 5 does not have this kind of adaptation. Anyway, in order to make the system more robust, we could easily add a restricted mode of the normal behavior zone representing the system running out of gas. The restricted zone should block the ignition although the presence of an incoming heat request. Once a quiescent state of the restricted zone is reached, the system should restore the normal functionalities by exhibiting undesired and then recovery behaviors.

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

IV. Zone-Based Mapping

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. Formally, we can characterize the active zone mapping with the following definition:

Definition 8: **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 elements in Z, such that:

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

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, considering the initial state S_0 of the gas burner example (Figure 5), $\Lambda(S_0)$ returns a set containing only the *normal* zone. In fact the enabled transitions in S_0 are *KeepIdle* and *Hreq*: both mapping into the normal behavior zone. Indeed, they represent normal events.

The *region* concept, informally introduced in section II-C (see Figure 4), can be now formally defined as follows:

Definition 9: **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 H, such that:

i $\forall S, S' \in H$ s.t. $S \neq S'$,
$\Lambda(S) = \Lambda(S') \wedge \Lambda(S)$ not restricted.

ii H locates in $TRG(PN_Z)$ a *weakly connected* [3] component.

Once the active zone mapping function is applied upon reachable states, we should identify the adaptation sets.

Definition 10: **Adaptation Set**. Given a zone-based TB net PN_Z and its state space $TRG(PN_Z)$ along with regions defined by the Λ function, an *adaptation set* is a set of $TRG(PN_Z)$ elements (both states and edges), identified by the following rules (one foreach timed adaptation model).

1) *Timed one-point adaptation set*: A one-point adaptation set is composed of a single edge e labeled with a cross-zone transition connecting two different regions.

2) *Timed overlap adaptation set*: An overlap adaptation set is composed of a set of states X and a set of edges Y, connecting two different regions. A state S belongs to X iff. $|\Lambda(S)| > 1$. Y contains all the outgoing edges departing from states in X, and the incoming edges of states in X departing from states outside X.

3) *Timed guided adaptation set*: An guided adaptation set is composed of a set of states X and a set of edges Y. A state S belongs to X iff. $z \in \Lambda(S)$ such that z is a restricted zone. Y is defined as described in the previous rule.

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

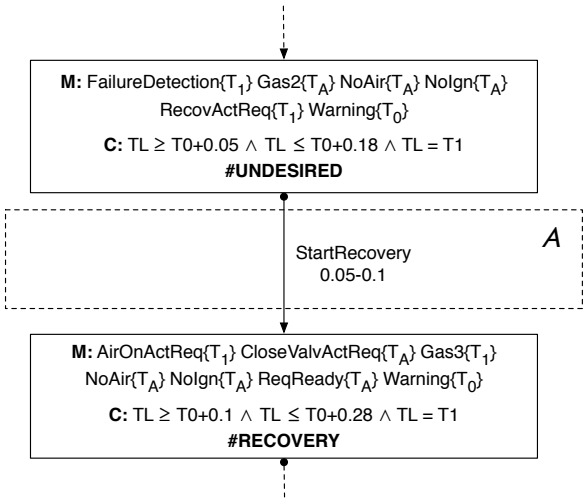


Fig. 6. Timed one-point adaptation set between the undesired region and the recovery region.

Figure 6 shows a timed one-point adaptation set. Two disjoint regions reifying the undesired and the recovery behaviors, respectively, are connected by a single edge labeled with the firing transition and temporal information. The symbol # precedes the name of the zones resulting from the evaluation of the Λ function. The two special variables T_L and T_A in symbolic states, represent respectively the last time-stamp produced and an *anonymous time-stamp* [4] not influencing the evolution of the model. Symbolic states are labeled with the name of their own region.

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 7 represents an example of timed overlap adaptation set. The adaptation set is composed of a set of states (S_9, S_{10}) and edges, labeled with the firing transitions and temporal information. Symbolic states in the adaptation set $\mathcal{A}$ belong to the two regions: normal behavior and undesired behavior, therefore the system exhibits multiple steady-state behaviors, i.e. the possibility to show multiple behavior at the same time.

The adaptation set $\mathcal{A}$ of a timed guided adaptation is composed of all the reachable states mapping into a restricted zone and all the edges connecting these states. The outgoing boundaries of $\mathcal{A}$ are made up by one-point adaptations leading from $\mathcal{A}$ into the target region.

V. Formal Verification

Based on the formal specification described in section III, we are able to verify the correctness of real-time self-adaptive systems by inspecting the TRG. In particular, we extended the GRAPHGEN software tool [4] (introduced in section II), in

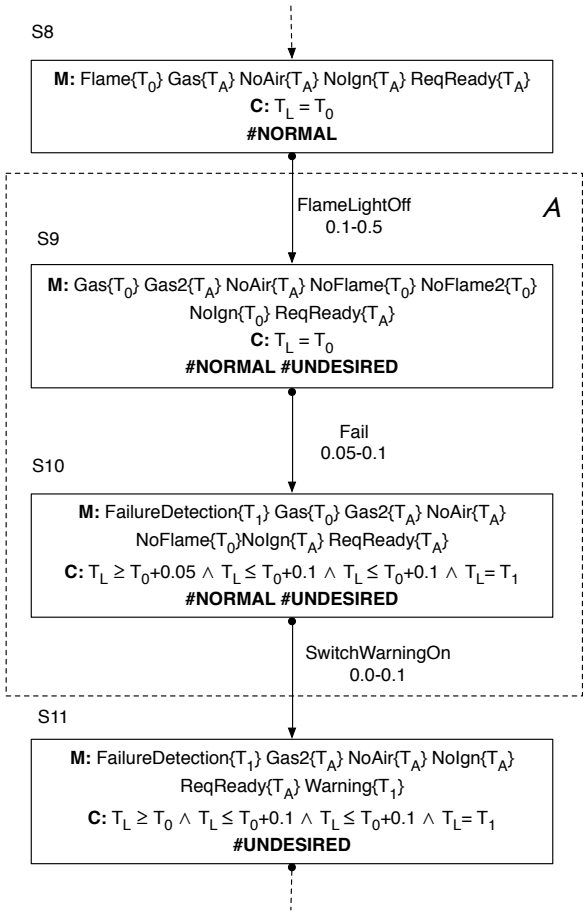


Fig. 7. Timed overlap adaptation set between the normal region and the undesired region.

order to support the verification of two categories of properties: *adaptation meta-properties* and *system-properties*.

A. Adaptation meta-properties

These general properties are related to adaptation and any self-* system (self-healing in our case) should satisfy them. They do not depend on the specific modeled system.

- *Cross-zone transition properties*. They aim at checking conformance of the generated *adaptation sets* w.r.t. the intended adaptation models. Foreach 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')
```

These commands, one for each adaptation model, are used to verify the structural properties described in section III-B. For instance, the *onepoint* 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 properties of interest, verified upon the gas burner model, are:

```
$ onepoint(#RECOVERY, BackNormal, #NORMAL)
$ overlap(#NORMAL, IgnFail, #UNDESIRED)
$ overlap(#NORMAL, FlameLightOff, #UNDESIRED)
```

- *Safety properties*. The system should never reach an invalid state, i.e., either a deadlock state or a state that belongs to the invalid behavior region. Therefore, foreach zone z, the following property should hold:

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

where "a ?> b" is used to verify that "whenever a holds, there exists a path leading to b". Some of the properties verified upon the gas burner model are:

```
$ !(#NORMAL ?> #INVALID)
$ !(#UNDESIRED ?> #INVALID)
$ !(#RECOVERY ?> #INVALID)
```

The name of a zone z preceded by # represents a condition satisfied by each reachable state mapping into z. The software tool verifies these properties through a TRG exploration, following the paths starting from the selected region. if for each state mapping into that region, does not exist a path leading into a state of the invalid region, the property holds.

- *Robustness properties*. Robustness properties are specific reachability properties used to verify that the system is always able to recover from a failure. These properties must be verified to ensure self-healing capability. The following robustness properties were verified upon the gas burner example:

```
$ #NORMAL ?> #UNDESIRED
$ #UNDESIRED −> #RECOVERY
$ #RECOVERY −> #NORMAL
```

where the "−>" operator is used to verify that "whenever a holds, eventually b will happen". This operator causes the software tool to perform a TRG exploration starting from the selected region a. If along the path either an invalid state or a loop is found before reaching the region b, the property does not hold. The example reported above prove the existence of a path leading from a normal state to an undesired state; for each undesired state, we always reach a recovery state; and for each recovery state, we always reach a normal state.

- *Timed properties*. The previous properties can be extended by adding temporal constraints. This is particularly useful to check whether adaptations are carried out within specific deadlines. For instance, we may ask if the gas burner can adapt itself from the normal behavior to the undesired behavior within 1.25 time units:

```
$ #NORMAL ?> #UNDESIRED, 1.25

TRUE
S6(#NORMAL) [0.1−0.5] S7(#NORMAL, #UNDESIRED)
[0.02−0.1] S10(#UNDESIRED)
```

The software tool starts by individuating each symbolic path σ connecting the normal and the undesired regions, where the sum of the maximum temporal distance values is less than 1.25. If this sufficient condition does not hold, the tool tries

to validate the property by looking for a concrete path in σ not violating the deadline.

Whenever the property holds, the program supplies a feasible path as a proof of correctness. The path contains information about regions associated with states and temporal information attached to edges (local minimum-maximum firing times). The example shows that the undesired state S_{10} is reachable from the normal state S_6 (within 1.25 time units), through S_7 that represents, along with the edges S_6-S_7 and S_7-S_{10}, an overlap adaptation set.

Other interesting timed properties, verified upon the gas burner, are for example:

```
$ #UNDESIRED −> #RECOVERY, 0.6
$ #RECOVERY −> #NORMAL, 1.1
$ #UNDESIRED −> #NORMAL, 1.7
```

This allows us to set up the temporal function associated with the *RecoveryFail* transition, in charge of bringing the system into an invalid state, if the recovery procedure fails.

B. System-properties

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

a) Intra-zone properties: These properties aim at verifying the correctness of zones in isolation.

In particular we can verify *invariant*, *safety*, or *liveness* properties. The verification can be performed by computing the TRG of a single zone of interest through the extended GRAPHGEN software tool.

• *Invariant properties*. Invariants should be preserved foreach 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 we verified the following invariant upon the normal behavior zone:

```
$ A(!Gas=1 || NoGas=0)
```

Meaning that, foreach reachable state (*A* operator), the presence of a token in the place *Gas* implies the absence of tokens in the place *NoGas*. The software tool verifies these properties by inspecting the symbolic marking associated with each reachable symbolic state.

• *Safety properties*. Safety properties are used to verify that "something bad will never happen". For instance we verified the following property upon the normal behavior zone:

```
$ !E(Idle=1 && Gas=1)
```

Meaning that do not exist a reachable state (*E* operator preceded by !) where both the place *Idle* and the place *Gas* are marked at the same time.

• *Liveness properties*. We can verify two different type of liveness properties, characterized by two different operators: the "a –> b" and "a ?> b" (introduced in section V-A), also adding temporal constraints.

For instance we verified the two following properties upon the normal behavior zone, and the recovery behavior zone:

```
$ IgnActReq=1 && ValActOpen=1 −> Gas=1 && Ignition=1, 0.6
$ CloseValvActReq=1 && AirOnActReq=1 −>
      Air=1 && NoGas2=1, 1.0
```

If the property does not hold, the program supplies a feasible path that breaks the property:

```
$ CloseValvActReq=1 && AirOnActReq=1 −>
      Air=1 && NoGas2=1, 0.9

FALSE
S22(#RECOVERY) [0.2−0.4] S25(#RECOVERY) [0.0−0.2]
S29(#RECOVERY)
```

The above example shows that both the gas flow interruption and the purging phase cannot be always achieved within 0.9 time units elapsed since the proper request. The counterexample shows a feasible path invalidating the property.

Another liveness property verified upon the normal behavior zone is reported below:

```
$ IgnActReq=1 && ValActOpen=1 ?> Flame=1, 0.7

TRUE
S13(#NORMAL) [0.0−0.05] S18(#NORMAL) [0.05−0.2]
S2(#NORMAL) [0.0−0.3] S4(#NORMAL) [0.02−0.1] S6(#NORMAL)
```

Since the property holds, the program returns a feasible path as a proof of correctness.

b) 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 model, through our extended GRAPHGEN software tool.

• *Invariant properties*. As an example, we show in the following some properties of interest we verified upon the gas burner:

```
$ A(!Gas=1 || (NoGas=0 && NoGas2=0))
$ A(!Flame=1 || (NoGas=0 && Gas2=0 && Gas3=0))
$ A(!Air=1 || (NoGas=0 && Gas2=0 && Gas3=0))
```

Foreach reachable state, we verified the proper mutual exclusion between specific marked places. For, instance, the *Gas* place should never be marked at the same time with *NoGas* and *NoGas2*.

• *Safety properties*. Some of the inter-zone safety properties verified upon the gas burner example are reported below:

```
$ !E(Idle=1 && (Gas=1 || Gas2=1 || Gas3=1))
$ !E(Gas=1 && Air=1)
$ !E(Flame=1 && FailureDetection=1)
$ !E(Flame=1 && FailureDetection=1)
```

For instance, the second property is used to prove that gas supply and air purge never happen 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. For instance, we verified that once the flame is up, it can fail reaching an undesired behavior:

```
$ Flame=1 ?> NoFlame2=1 && Gas2=1
```

Moreover, we verified that once a failure is detected, the system exhibits proper recovery behavior within 0.4 time units:

```
$ FailureDetection=1 −>
      CloseValvActReq=1 && AirOnActReq=1, 0.4
```

VI. Related Work

The technique presented in this paper has been mainly influenced by different related works on formal specification and verification of self-adaptive systems. [28] proposes Petri Nets modeling and LTL (*Linear Temporal Logic*) model checking, in order to verify correctness of adaptations and robustness properties. Here we extended the adaptation models – *one-point adaptation*, *overlap adaptation*, and *guided adaptation* – originally presented in [27] and instantiated using Petri Nets in [28]. In particular we have taken into account concerns associated with time, by including 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, departing from [28], we instantiated the extended adaptation models with TB nets, that represent a very expressive formal model for describing rea-time or even time-critical systems. Moreover we 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. The identification of different behaviors upon the state space structure was originally introduced in [16]. We introduced a technique able to map zones defined upon the model into regions of the state space.

Other Petri Net based approaches to model and verify dynamically adapting programs have been recently introduced. [9] proposed a formal method to specify self-adaptive systems, that allows users to reason about the correctness of the system in spite of its dynamic reconfigurations. The proposed programming model is coupled with a programming language realizing the Context-Oriented Programming (COP) paradigm [25]. COP languages allow behavioral adaptations to be dynamically added and removed at run time. [8] supplies a reflective framework that lets users model a system able to adapt itself, keeping separated functional aspects from adaptation ones and applying changes to the model if necessary. A timed stochastic semantics for this approach is provided in [7]. Anyway, these two techniques are not effective when dealing with real-time constraints, because quantitative temporal aspects cannot be handled.

Our analysis technique has been implemented by extending the GRAPHGEN software tool, introduced in [4]. Our extended version can be used to verify correctness properties upon either any zone in isolation, or upon the entire system to verify the correctness of adaptation behaviors. Departing from [28], we grant the possibility of verifying the conformance of the behavior against intended adaptation models by means of *cross-zone transition properties*; we support the verification of *timed properties* in order to check real-time constraints; and we support the verification of *robustness properties* to verify time constrained self-healing behaviors.

Concerning fault-tolerance and self-healing capability analysis, [14] presents a case study in formal modeling and verification of a robotic system that deals with failures. Modeling is performed using a transition automata and correctness is checked using LTL and CTL (*Computational Tree Logic*). [20] outlines an approach for modeling and analyzing fault tolerance and self-adaptive mechanisms in distributed systems. It proposes a modal action logic formalism to describe normal and abnormal (undesired) behavior. Anyway all these works deal with time only in a qualitative way, thus making those approaches ineffective to verify correctness of real-time self-adaptive systems.

Complementary approaches can be followed to ensure correctness at run-time. [17] models the MAPE-K feedback loop (i.e., the model of the knowledge and the adaptation components [11]) using networks of timed automata [5] and evaluates the approach 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 increments with time steps. Foreach 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 will then execute tasks associated with these invalidated nodes in non-deterministic order. 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.

VII. Conclusion

This paper introduces the *zone-based* technique to specify and verify the behavior of dynamically adaptive real-time systems. We extended the adaptation models introduced in [28], with temporal constraints and different temporal semantics, in order to model both mandatory and optional timed events. 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 that the entire system satisfies *inter-zone properties*. The verification of temporal aspects is supported through *timed properties*, able to check that both functional aspects and adaptations comply with specific temporal deadlines. Moreover, we added the possibility of verifying 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.

We have shown the effectiveness of our approach by modeling and verifying a real-time self-adaptive Gas Burner system proving correctness of both functional, adaptation, self-healing, and temporal aspects.

Acknowledgment

This work has been partially supported by the Italian Ministry of Research within the PRIN project GENDATA 2020 and by the Lombardy Region SMART BREAK project.

REFERENCES

[1] Robert Allen, Rmi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998.

[2] Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Symbolic state space exploration of RT systems in the cloud. In *Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC 2012, pages 295–302, Los Alamitos, CA, USA, 2012. IEEE CS Press.

[3] Carlo Bellettini, Matteo Camilli, Lorenzo Capra, and Mattia Monga. Mardigras: Simplified building of reachability graphs on large clusters. In ParoshAziz Abdulla and Igor Potapov, editors, *Reachability Problems*, volume 8169 of *LNCS*, pages 83–95. Springer Berlin Heidelberg, 2013.

[4] Carlo Bellettini and Lorenzo Capra. Reachability analysis of time basic Petri nets: A time coverage approach. *2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 0:110–117, 2011.

[5] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jrg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg, 2004.

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

[7] Lorenzo Capra. A symbolic reachability graph and associated markov process for a class of dynamic petri nets. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 458–461, Aug 2010.

[8] Lorenzo Capra and Walter Cazzola. A petri-net based reflective framework for the evolution of dynamic systems. *Electronic Notes in Theoretical Computer Science*, 159(0):41 – 59, 2006. Proceedings of the First {IPM} International Workshop on Foundations of Software Engineering (FSEN 2005) Foundations of Software Engineering 2005.

[9] N. Cardozo, S. Gonzalez, K. Mens, R. Van Der Straeten, and T. DHondt. Modeling and analyzing self-adaptive systems with context petri nets. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, pages 191–198, July 2013.

[10] Betty HC Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.

[11] Rogério de Lemos and José Luiz Fiadeiro. An architectural support for self-adaptive software for treating faults. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 39–42, New York, NY, USA, 2002. ACM.

[12] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[13] Guillaume Gardey, Olivier H. Roux, and Olivier F. Roux. State space computation and analysis of time petri nets. *Theory Pract. Log. Program.*, 6(3):301–320, May 2006.

[14] Matthias Gdemann, Frank Ortmeier, and Wolfgang Reif. Formal modeling and verification of systems with self-x properties. In LaurenceT. Yang, Hai Jin, Jianhua Ma, and Theo Ungerer, editors, *Autonomic and Trusted Computing*, volume 4158 of *Lecture Notes in Computer Science*, pages 38–47. Springer Berlin Heidelberg, 2006.

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

[16] M. Usman Iftikhar and Danny Weyns. A case study on formal verification of self-adaptive behaviors in a decentralized system. In Natallia Kokash and António Ravara, editors, *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012.*, volume 91 of *EPTCS*, pages 45–62, 2012.

[17] M. Usman Iftikhar and Danny 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, pages 125–134, New York, NY, USA, 2014. ACM.

[18] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[19] Woo Jin Lee, Sung Deok Cha, and Yong Rae Kwon. Integration and analysis of use cases using modular petri nets in requirements engineering. *Software Engineering, IEEE Transactions on*, 24(12):1115–1130, Dec 1998.

[20] Jeff Magee and Tom Maibaum. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*, SEAMS '06, pages 30–36, New York, NY, USA, 2006. ACM.

[21] Ernst-Rdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[22] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[23] Paul Robertson and Robert Laddaga. Model based diagnosis and contexts in self adaptive software. In Ozalp Babaoglu, Mrk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen, editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 112–127. Springer Berlin Heidelberg, 2005.

[24] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[25] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 85(8):1801–1817, August 2012.

[26] Danny Weyns, M. Usman Iftikhar, Didac Gil de la Iglesia, and Tanvir Ahmad. A survey of formal methods in self-adaptive systems. In Bipin C. Desai, Emil Vassev, Sudhir P. Mudur, and Bipin C. Desai, editors, *C3S2E*, pages 67–79. ACM, 2012.

[27] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[28] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proc. of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.