VTS-based Specification and Verification of Behavioral Properties of AADL Models[⋆]

D. Monteverde[1,3], A. Olivero[1], S. Yovine[2⋆⋆], and V. Braberman[3⋆⋆⋆]

[1] Inst. de Tecnología, UADE, Argentina. {damonteverde|aolivero}@uade.edu.ar
[2] VERIMAG-CNRS, France. Sergio.Yovine@imag.fr
[3] Departamento de Computación, FCEyN, UBA, Argentina. vbraber@dc.uba.ar

Abstract. AADL is an aerospace standard for model-driven design of complex real-time embedded systems. Currently, behavioral properties of AADL models can be specified inside the system description using AADL concepts or outside it using external textual languages, and verified using schedulability analysis or (Time Petri Net-based) model-checking tools. This paper (1) proposes Visual Timed Scenarios (*VTS*) as a graphical property specification language for AADL, and (2) devises an effective translation from *VTS* to Time Petri Nets (TPN) which enables model-checking properties expressed in *VTS* over AADL models using TPN-based tools integrated into AADL-compliant IDEs (e.g., TOPCASED).

1 Introduction

The Architecture Analysis and Design Language (AADL) [13] is an aerospace standard released by the Society of Automotive Engineers (SAE) for model-based specification and analysis of complex real-time embedded systems. AADL has been designed to support model-based and formal analyses of critical properties. For this, AADL provides modeling concepts for the description of application system architectures in terms of suitable abstractions of software and hardware components and the interactions between them. The definition of AADL motivated the development of AADL-centric tools such as OSATE[4] and Ocarina [10], as well as the integration of AADL into domain-specific model-driven software engineering environments, such as TOPCASED[5]. This enabled different kinds of formal analyses, including schedulability, e.g., with Cheddar [14], and model-checking, e.g., with Time Petri Net-based tools like Tina [4] and Romeo [9].

A way of enhancing the usability of formal techniques in model-driven system design and flows analysis consists in resorting to visual languages capable of representing and visually presenting application semantics in a clear, precise way, specially in the context of event-based systems. Following this idea, in this

⋆ Partially supported by ANPCyT projects PICT 32440 and PICTO-CRUP 31352, STIC-AmSud project TAPIOCA and UBACyT X021
⋆⋆ Currently visiting UBA and UADE.
⋆⋆⋆ Conicet
[4] http://www.aadl.info/
[5] http://www.topcased.org

paper we adopt Visual Timed Scenarios (VTS) [1] as a language for specifying behavioral properties of models of systems described in AADL. In order to make possible the verification of properties expressed in VTS over AADL models using available tools integrated into AADL-complaint IDEs, we devise a translation from VTS to Time Petri Nets. Once a visually specified property is translated into TPN, existing model checking tools can be used to verify if operational descriptions encompassed by AADL model satisfy the expected property. Fig. 1 exhibits the integration of the AADL models with VTS scenarios in a tool chain. The part concerning VTS (enclosed in gray) will be explained along this work.

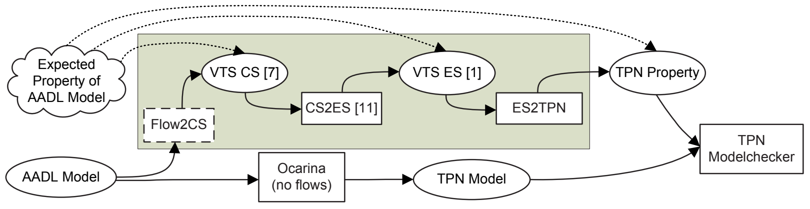


Fig. 1. Tool chain integrating AADL and VTS

The paper is structured as follows. Sec. 2 recalls VTS by means of an example. Sec. 3 briefly reviews Time Petri Nets (TPN). Sec. 4 presents the translation of VTS into TPN. Sec. 5 proposes a procedure to model-check whether a TPN satisfies a property expressed in VTS. Sec. 6 illustrates the application of these results for verifying different behavioral properties of AADL models: (1) mode-change behaviors, and (2) flow specifications.

2 Visual Timed Scenarios (VTS)

2.1 Informal presentation

Visual Timed Scenarios [1, 7] language is used to describe *event patterns*, which can be regarded as simple, graphical depictions of predicates over traces (time-stamped executions), constraining expected behavior. It basically features annotated partial order of relevant events, denoting a (possibly infinite) set of matching traces. Violation of verification goals for models such as freshness, bounded response or event correlation can naturally be expressed using the notation.

The basic elements of VTS graphical notation are points connected by lines and arrows. Points are labeled by sets of events, meaning that the point stands for an occurrence of one of the events in an execution. VTS can represent *precedence relations* and *temporal distances* between points; and sets of events which are *forbidden* between them. The detailed formalization of VTS and its thorough comparison with other visual languages is given in [7]. Here, we informally introduce VTS through a simple, yet illustrative, example.

Consider a system composed of two jobs Job_1 and Job_2 (Fig. 2, based on [2]). The behavior of the system is as follows: (1) Job_1 if started, always terminates; (2) Job_2 if started, always terminates; (3) Job_2 can not start while Job_1 is in execution; (4) Job_1 must terminate in at most 12; (5) Job_2 must wait at least 14 to start; (6) The temporal distance between both jobs' ends is at most 10.

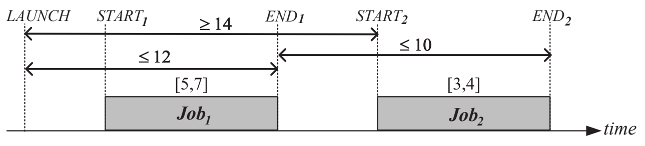


Fig. 2. Example of two jobs

Fig. 3 illustrates these requirements expressed in VTS as *conditional scenarios* [7]. Conditional scenarios allow to state that whenever an *antecedent* sub-scenario (depicted in black) happens, a *consequent* sub-scenario (depicted in gray) must happen too.

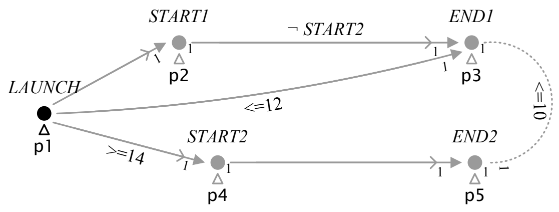


Fig. 3. VTS Conditional scenarios for requirements of two jobs example

Points are labeled with *events*. Triangles below points are used to display optional point names, needed for the formal notation. An arrow between two points specifies a *precedence* relationship. Arrow labels specify *forbidden events* between points: for instance, there is no *START2* event between *START1* and *END1*. A double forward arrow means "the next" occurrence of the event of the destination point (i.e., shorthand for labeling the arrow with the destination's label). A double backward arrow meas "the previous" occurrence of the event of the source point (i.e., shorthand for labeling the arrow with the source's label). A dashed line linking two points expresses a *temporal distance* between them. Dashed lines can also be labeled with forbidden events. Fig. 4 shows the graphical notation of VTS elements used in this work[6].

Verifying conditional scenarios is done by building (using the CS2ES tool showed in Fig. 1) a set of *existential* scenarios that stand for all possible counterexamples of the conditional scenarios [7, 11]. These scenarios, a.k.a. anti-scenarios, model all the ways in which a conditional scenario may be violated by the system. This work only relies on how to model-check existential scenarios, and therefore, hereinafter, existential scenarios are referred as "scenarios". Fig. 5 illustrates all the VTS anti-scenarios of the conditional scenario of Fig. 3.

[6] VTS has more primitives, that increase its expressive power, which are omitted here for the sake of simplicity. The interested reader is referred to [7].

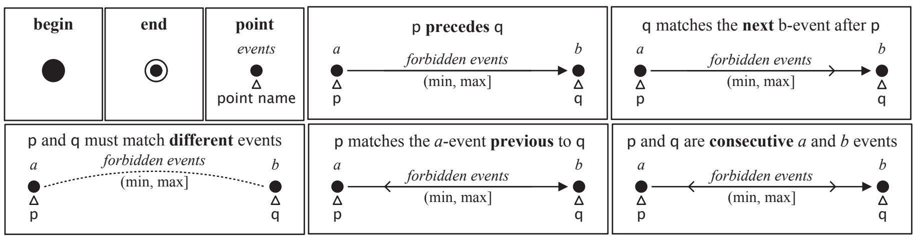
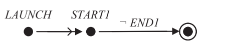
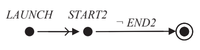

begin
end
point
events
point name
p precedes q
a
forbidden events
(min, max]
b
p
q
q matches the next b-event after p
p and q must match different events
p matches the a-event previous to q
p and q are consecutive a and b events

Fig. 4. VTS graphical notation

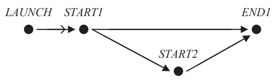
A big full circle stands for the *begin* of the execution, and two concentric circles correspond to its *end*.



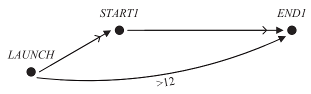
(a) Job_1 starts, but does not terminate (1)



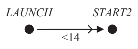
(b) Job_2 starts, but does not terminate (2)



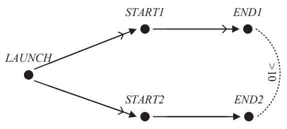
(c) Job_2 starts while Job_1 is in execution (3)



(d) Job_1 terminates after 12. (4)



(e) Job_2 starts before 14. (5)



(f) More than 10 passes between jobs ends. (6)

Fig. 5. Anti-scenarios (existential scenarios).

2.2 Formal presentation

Definition 1 **(VTS syntax).** *A* scenario *is a tuple* $\langle \Sigma, P, \ell, \not\equiv, <, \gamma, \delta \rangle$ *, where:*

- Σ *is a finite set of* events*;*
- P *is a finite set of* points*;*
- $\ell : P \to 2^{\Sigma}$ *is labels each point with a non-empty set of events;*
- $\not\equiv\ \subseteq P \times P$ *is an asymmetric relation (*inequality*) between points (graphically represented by* dotted *lines);*
- $< \subseteq (P \uplus \{\mathbf{0}\} \times P \uplus \{\infty\}) \smallsetminus \{\langle \mathbf{0}, \infty \rangle\}$ *is a* precedence *relation between points (graphically represented by* arrows*), where* $\mathbf{0}$ *and* ∞ *represent the* begin *and the* end *of an execution, resp.;*
- $\gamma : (\not\equiv \cup <) \to 2^{\Sigma}$ *assigns to each pair of points, related by inequality or precedence, the set of events* forbidden *between them;*
- $\delta : (\not\equiv \cup (< \smallsetminus (P \times \{\infty\}))) \to \mathfrak{I}$ *assigns to each inequality or precedence relation an integer-bounded or upper-unbounded interval of non-negative real-numbers restricting the* time elapsed *between the two points.*

Given a set C, a *sequence over* C is a (possibly infinite) sequence of elements from C. Given a sequence s, $|s|$ is its length ($|s| \overset{def}{=} \infty$ when s is infinite) and

$\Pi(s) \overset{def}{=} \{i \in \mathbb{N} \;/\; 0 \leq i < |s|\}$ is the set of positions of s. Given $i, j \in \Pi(s)$, s_i is the i^{th} element of s; $s_{i]}$ is the prefix ending at position i; $s_{[i}$ is the suffix starting at position i and $s_{[i,j]}$ is the sub-sequence from position i to position j (if $i > j$, $s_{[i,j]} \overset{def}{=} s_{[j,i]}$). Using '(' or ')' instead of '[' or ']' means the corresponding sub-sequence does not include its border(s). We call $first(s)$ the first element of s. If s is finite, $last(s)$ is its last element. For $X \subseteq C$, $s \cap X$ denotes the set of elements of X appearing in s, i.e., $\{x \in X \mid \exists i.\; s_i = x\}$.

A *temporal sequence* is a weakly increasing sequence of timestamps (i.e., non negative real numbers). Given a finite temporal sequence τ we define $\Delta(\tau)$ as the time elapsed during τ: $\Delta(\tau) = last(\tau) - first(\tau)$ or 0 if $|\tau| = 0$. A temporal sequence τ can be *shifted* by a real number ϵ producing a temporal sequence called $\tau + \epsilon$, such that $\forall i \in \Pi(\tau); (\tau + \epsilon)_i = \tau_i + \epsilon$.

A *trace* over a set C is a pair $\langle s, \tau \rangle$ where s is a sequence over C and τ is a temporal sequence of the same length. Given a trace $\sigma = \langle s, \tau \rangle$, we define $|\sigma| \overset{def}{=} |s|$ and $\Pi(\sigma) \overset{def}{=} \Pi(s)$. A trace is *time-divergent* iff for any real number T there exists a position k such that $\Delta(\tau_{k]}) > T$.

The semantics of VTS assigns to each scenario a set of traces satisfying it. Labeled points represent events in the traces, they can match a particular position in a trace if the event in that position is among the allowed events associated to the point by the labeling function ℓ.

Intuitively, a *matching* is a mapping between points in a scenario and positions in a trace, exhibiting how the trace satisfies the scenario. Formally:

Definition 2 (VTS semantics). *Given a scenario $\mathcal{S} = \langle \Sigma_{\mathcal{S}}, P, \ell, \not\equiv, <, \gamma, \delta \rangle$, a trace $\sigma = \langle s, \tau \rangle$ over Σ' where $\Sigma_{\mathcal{S}} \subseteq \Sigma'$, and a mapping $\hat{\cdot} : P \to \Pi(\sigma)$; we say that $\hat{\cdot}$ is a* matching *between $\mathcal{S}$ and σ iff for all points $\mathsf{p}, \mathsf{q} \in P$:*

M1 *$s_{\hat{\mathsf{p}}} \in \ell(\mathsf{p})$; the mapping for a point is a position of the trace with an event that labels this point.*

M2 *if $\mathsf{p} \not\equiv \mathsf{q}$ then $\hat{\mathsf{p}} \neq \hat{\mathsf{q}}$; two different points cannot map to the same position.*

M3 *if $\mathsf{p} < \mathsf{q}$ then $\hat{\mathsf{p}} < \hat{\mathsf{q}}$; the position of the source point must be smaller than the destination's.*

M4 *$s_{(\hat{\mathsf{p}},\, \hat{\mathsf{q}})} \cap \gamma(\mathsf{p}, \mathsf{q}) = \emptyset$; no forbidden event can appear in the sub-trace defined by corresponding occurrences of the points.*

M5 *$s_{\hat{\mathsf{p}})} \cap \gamma(\mathbf{0}, \mathsf{p}) = s_{(\hat{\mathsf{p}}} \cap \gamma(\mathsf{p}, \infty) = \emptyset$; no forbidden event specified between begin (resp., a point) and a point (resp., end) can appear before (resp., after) the corresponding occurrence of the point.*

M6 *$\Delta(\tau_{[\hat{\mathsf{p}},\, \hat{\mathsf{q}}]}) \vDash \delta(\mathsf{p}, \mathsf{q})$; the time elapsed between the occurrences of the corresponding points must satisfy the specified time restriction.*

M7 *$\Delta(\tau_{\hat{\mathsf{p}}]}) \vDash \delta(\mathbf{0}, \mathsf{p})$; the time elapsed since begin until the occurrence of a point must satisfy the specified time restriction.*

*Rules **M4-5** and **M6-7** must be considered only when the domains of the functions γ and δ are defined, respectively.*

Definition 3 (Existential VTS Semantics). *We say that a trace σ* satisfies *a scenario $\mathcal{S}$ (noted $\sigma \vDash \mathcal{S}$) iff there exists at least one matching between them.*

3 Time Petri Nets (*TPNs*)

Time Petri Nets [5] are a widely used formalism for timed systems. They are supported by several tools (e.g. TINA [4], Romeo [9]). TPNs extend Petri nets with temporal intervals associated with transitions: assuming transition t, with an interval $[\alpha, \beta]$, became last enabled at time τ, then t cannot fire earlier than time $\tau + \alpha$ and must fire no later than $\tau + \beta$, unless disabled by firing some other transition. Firing a transition takes no time.

3.1 TPNs Formal Syntax

Definition 4 (Time Petri Net). *A* Time Petri Net[7] *is a tuple* $\mathcal{N} = \langle \boldsymbol{S}, \boldsymbol{T}, \boldsymbol{Pre}, \boldsymbol{Post}, \Sigma_{\mathcal{N}}, \boldsymbol{L}, \boldsymbol{Inh}, \succ, m^0, I^s \rangle$*, where:*

- $\boldsymbol{S}$ *is a finite set of places.*
- $\boldsymbol{T}$ *is a finite set of transitions.*
- $\boldsymbol{Pre} \subseteq \boldsymbol{T} \times \boldsymbol{S}$ *is a relation between transitions and input places.*
- $\boldsymbol{Post} \subseteq \boldsymbol{T} \times \boldsymbol{S}$ *is a relation between transitions and output places.*
- $\Sigma_{\mathcal{N}}$ *is a finite set of events.*
- $\boldsymbol{L} : \boldsymbol{T} \to \Sigma_{\mathcal{N}} \cup \{\lambda\}$ *is a function that labels each transition with an event or with* $\lambda \notin \Sigma_{\mathcal{N}}$*. We assume that* $\forall\ e \in \Sigma_{\mathcal{N}}$*,* $\exists\ t \in \boldsymbol{T}$*, s.t.* $\boldsymbol{L}(t) = e$*.*
- $\boldsymbol{Inh} \subseteq \boldsymbol{T} \times \boldsymbol{S}$ *is a relation that defines inhibitor places for transitions.*
- $\succ\ \subseteq \boldsymbol{T} \times \boldsymbol{T}$ *is a priority (irreflexive, asymmetric, and transitive) relation between transitions.*
- $m^0 \subseteq \boldsymbol{S}$ *is a set of places with initial marking.*
- $I^s : \boldsymbol{T} \to \mathfrak{I}$ *is a function called* static interval *function.*

Fig. 6 summarizes the graphical notation for TPNs used in this work.

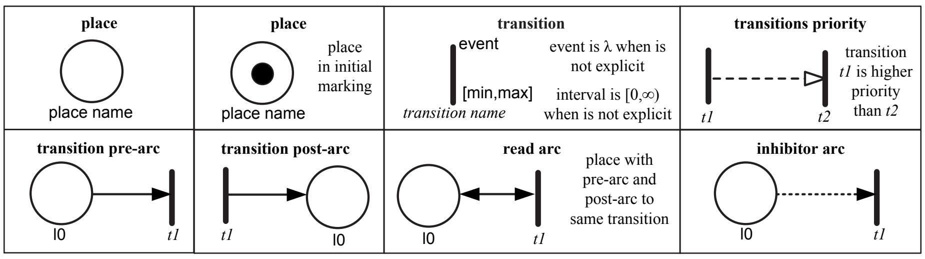


Fig. 6. TPN graphical notation

Parallel Composition. This operation combines two TPNs in one TPN where transitions with the same label (different from λ) are merged. The parallel composition between two TPNs, $\mathcal{N}_1$ and $\mathcal{N}_2$, is denoted as $\mathcal{N}_1 \| \mathcal{N}_2$. See [5] for more details.

[7] For simplicity, we consider here ordinary (i.e. all arcs have weight 1) TPNs, but the results can be extended to non-ordinary ones.

3.2 TPNs Semantics

Given a TPN $\mathcal{N}$, a *state* of $\mathcal{N}$ is a pair $\omega = \langle m, I \rangle$, where $m : \mathbf{S} \rightarrow \mathbb{N}$ is a marking and $I : \mathbf{T} \rightarrow \mathfrak{I}$ is the interval function that associates a temporal interval with every transition enabled at m. The initial state is denoted ω_0.

The semantics of TPNs defines the evolution of a TPN state resulting from the firing of transitions and passage of time. The reader is referred to [5] for the detailed semantics.

We write $\omega \xrightarrow{\mathbf{L}(t)@\theta} \omega'$ to denote that from state ω, transition t is fired after a time θ, resulting in state ω'; and $\omega \xrightarrow{\lambda@\theta} \omega'$ to denote that from state ω, time can elapse to state ω'. An *execution* is a time-divergent sequence $\rho : \omega_0 \xrightarrow{a_0@\theta_0} \omega_1 \xrightarrow{a_1@\theta_1} \ldots$ We write m_{ρ_i} to denote the marking of the i-th state of ρ. The time-divergent trace of ρ is $\sigma = \langle s, \tau \rangle$ with $s = a_0, a_1 \ldots$, and $\tau = \vartheta_0, \vartheta_1 \ldots$, where $\vartheta_0 = \theta_0$ and $\vartheta_i = \vartheta_{i-1} + \theta_i$, for $i \geq 1$.

4 Translating VTS into TPN

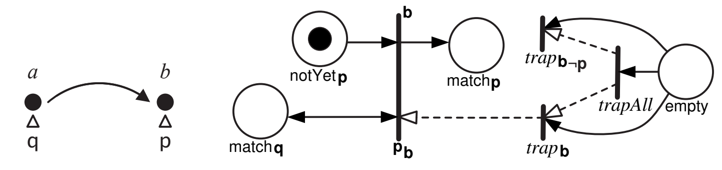
The translation algorithm proceeds as follows: for each part of the VTS scenario that must be matched, it builds a TPN component. So, each point, forbidden event, time restriction, precedence between points, etc., in the VTS scenario, generates a TPN. The translation of the whole scenario is obtained by the *fusion* (a special composition, see below) of all components. The rules that formally define this translation can be found in [12]. The ES2TPN tool (Fig. 1) performs this whole process.

4.1 Construction of TPN components

Construction of TPN components for matching points. In order to recognize occurrences of events as matchings of points, we construct a TPN as follows. For every point p of the VTS scenario, we add two places to the TPN: $\mathsf{notYet_p}$ and $\mathsf{match_p}$. The place $\mathsf{notYet_p}$ has an initial marking and represents that no event labeling point p has occurred yet. The place $\mathsf{match_p}$, if marked, models that a matching event for this point has occurred. Between these places, we add the possible matching transitions: one transition for each event e labeling point p. Each of these is labeled with e, and has a *pre-arc* from $\mathsf{notYet_p}$ and a *post-arc* to $\mathsf{match_p}$. Also, we must consider the case where two (or more) points match the same event. Therefore, we add transitions for all possible combinations of multiple matching points for each event.

To take into account precedence relations among points, for every matching transition into place $\mathsf{match_p}$ we add a *read-arc* from any place $\mathsf{match_q}$, whenever there is a precedence arrow from q to p (this is because place $\mathsf{match_q}$ must be marked before marking place $\mathsf{match_p}$).

Finally, this component has special transitions which will be used in the construction of forthcoming components. Transition $trap_e$ is set with higher priority than any matching transition labeled with event e. Transition $trapAll$ has higher priority than all transitions labeled $trap_e$, and therefore higher than all matching transitions (by transitivity). For every point p and event e labeling p, a transition $trap_{e\neg \mathsf{p}}$ is added, with higher priority than any transition matching event e but not matching point p. The purpose of these transitions is to define a priority schema, not to be fired, as they are always disabled by adding a *pre-arc* from a place called empty which is never marked. Fig. 7 gives an example of the construction of TPN component for matching points.

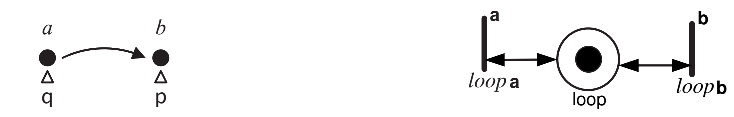


(a) *VTS* scenario (b) TPN component for point p

Fig. 7. TPN component construction for *matching points.*

Construction of TPN components for events not matched by any point. To recognize occurrences of events not associated to point matchings, we add a unique place loop, with an initial marking, and loop transitions for each event e of the scenario.

Fig. 8 shows the resulting TPN for a simple example.



(a) *VTS* scenario (b) TPN of unmatched occurrences of events a and b

Fig. 8. TPN component construction for *unmatched events.*

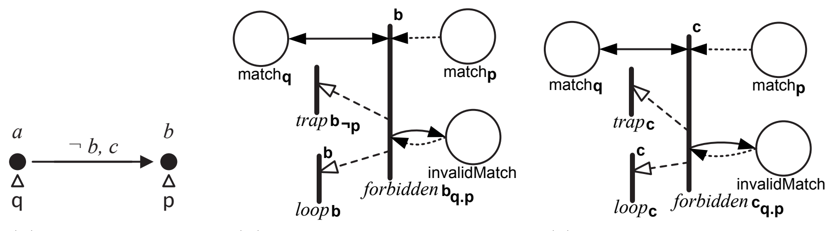
Construction of components for forbidden events on precedence relations. Suppose there is precedence relation from point q to p, and let $\mathsf{match}_\mathsf{q}$ and $\mathsf{match}_\mathsf{p}$ be the corresponding matching places of the points. For each forbidden event e on the precedence relation, a forbidden transition, labeled with e, is added with a *pre-arc* from $\mathsf{match}_\mathsf{q}$ and an *inhibitor-arc* from $\mathsf{match}_\mathsf{p}$.

In order for this transition to capture all possible occurrences of the forbidden event e, if e is labeling p, a priority relation is added to transition $trap_e$, otherwise is added to transition $trap_{e\neg \mathsf{p}}$. As we have seen, $trap_e$ has higher priority than any matching transition for event e, and $trap_{e\neg \mathsf{p}}$, has higher priority than any matching transition for event e not related with p.

Also, the corresponding loop transition for event e is disallowed whenever the forbidden transition is enabled, by setting a priority relation. Therefore, the loop transition is enabled only when point q has occurred but not yet point p, avoiding any occurrence of event e not corresponding to the matching point p.

At last, a *post-arc* with an *inhibitor-arc* is added to place invalidMatch. This place, as we will show later, if not empty, avoids reaching the acceptance condition for matching the whole VTS scenario. The purpose of this *inhibitor-arc* is to ensure the boundedness of the TPN.

Fig. 9 illustrates the construction of TPN components for *forbidden events on precedence relations*.



(a) VTS scenario (b) TPN for forb. event b (c) TPN for forb. event c

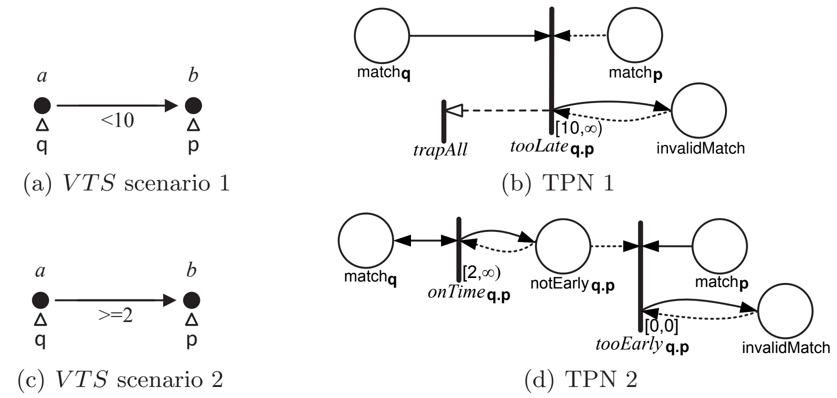
Fig. 9. TPN components for *forbidden events over precedence relations.*

Construction of TPN components for temporal restrictions on precedence relations. In VTS, temporal restrictions over a precedence relation can involve two cases: (1) when the time elapsed between the source and destination points has a maximum allowed value, and (2) when it has a minimum allowed value. Note that VTS time restrictions allow both cases to be combined in an interval constraint.

In case of an upper limit β, we add a transition $tooLate_{\mathsf{q \cdot p}}$ with a lower bound of β. This transition has a *read-arc* from place $\mathsf{match_q}$, an *inhibitor-arc* from place $\mathsf{match_p}$, and a *post-arc* with an *inhibitor-arc* to place invalidMatch. We add a priority relation from this transition to *trapAll* to avoid matching points when it is enabled. Therefore, this transition will avoid point p to match after a time β since point q has occurred. Fig. 10(a) and 10(b) illustrates this construction.

In case of a lower limit α, we use two transitions. One transition, called $onTime_{\mathsf{q \cdot p}}$, will delay at least α after point q matches, leaving a token at a new place $\mathsf{notEarly_{q \cdot p}}$. The other transition, called $tooEarly_{\mathsf{q \cdot p}}$, has a *pre-arc* from place $\mathsf{match_p}$, an *inhibitor-arc* from place $\mathsf{notEarly_{q \cdot p}}$, and a *post-arc* with an *inhibitor-arc* to place invalidMatch. Therefore, this transition will prevent a scenario matching if point p occurs, but not transition $onTime_{\mathsf{q \cdot p}}$ which only becomes enabled after a time α since point q's occurrence. Fig. 10(c) and 10(d) illustrates this construction.

Construction of TPN components for restrictions over inequality relations. Consider two points q and p, such that $\mathsf{p} \not\equiv \mathsf{q}$. By definition these points have different matching, then necessarily, either q occurs before p, or p occurs before q. Therefore, both cases must be considered. For this, we apply the rules explained above for taking care of precedence relations.

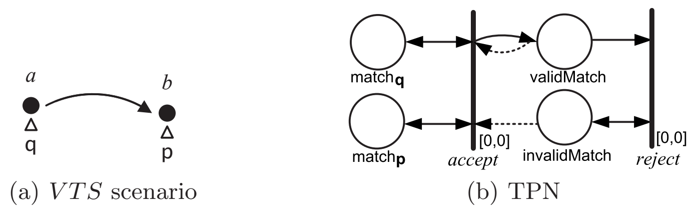


(a) *VTS* scenario 1 (b) TPN 1 (c) *VTS* scenario 2 (d) TPN 2

Fig. 10. TPN components for *time restrictions over precedence relations*

4.2 Construction of the TPN for the whole scenario

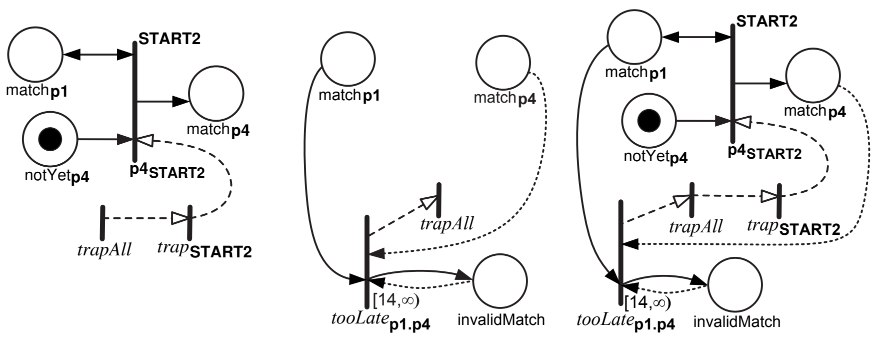
Scenario matching We add a place, namely, validMatch, and two transitions, namely, *accept* and *reject*. Transition *accept*, immediately fires if all points have been matched, and only if place invalidMatch is empty, putting a token in validMatch. Transition *reject*, fires as soon as invalidMatch is reached, removing all tokens (if any) from validMatch. This transition is needed to wait for occurrences of forbidden events in the future. Fig. 11 illustrates this construction.



(a) *VTS* scenario (b) TPN

Fig. 11. TPN component for *scenario matching*.

Fusion of TPNs. Now, we introduce the *fusion* operation, to obtain a TPN by combining two or more TPNs. This operation is based on set union; so if two combined TPNs share places and transitions, these will appear once in the final construction. The fusion operation between two TPNs, $\mathcal{N}_1$ and $\mathcal{N}_2$, is denoted as $\mathcal{N}_1 \oplus \mathcal{N}_2$. Fig. 12 illustrate fusion operation. Resulting fusion of TPNs Fig. 12(a) and Fig. 12(b) is presented in Fig. 12(c).



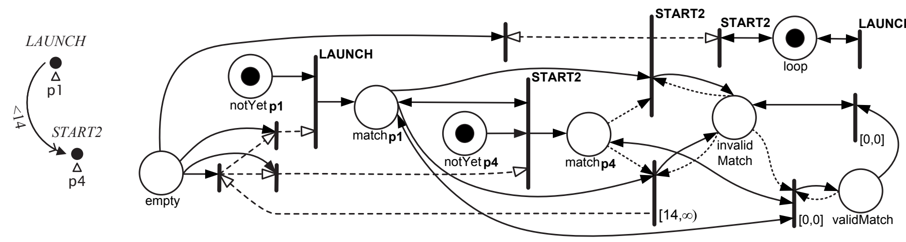
(a) TPN $\mathcal{N}_1$ (b) TPN $\mathcal{N}_2$ (c) TPN $\mathcal{N}_1 \oplus \mathcal{N}_2$

Fig. 12. TPNs' fusion sample

Definition 5. *Given a scenario $\mathcal{S}$, we define the TPN of $\mathcal{S}$, denoted $\mathcal{T}_{\mathcal{S}}$, as the fusion of the component TPNs constructed as explained above.*

Example. Fig. 13 shows the resulting TPN for the VTS scenario initially presented at Fig. 5(e)[8]. For this scenario, the TPN results from the fusion of the following components:

- **Matching points**: for points p4 (Fig. 12(a)) and p1.
- **Unmatched event**: for events $START2$ and $LAUNCH$.
- **Forbidden events**: for the forbidden event of $START2$ labeling the precedence relation from point p1 to p4.
- **Temporal restrictions**: for time restriction of < 14 labeling the precedence relation from point p1 to p4 (Fig. 12(b)).
- **Scenario Matching**.



(a) VTS (b) TPN

Fig. 13. TPN for scenario: Job_2 starts before 14.

[8] In Fig. 13(b) transition names have been omitted in order to keep the figure small and readable.

5 Model Checking *VTS*

The problem of checking whether a system under analysis (SUA) modeled as a TPN $\mathcal{N}$ satisfies a VTS scenario $\mathcal{S}$ is solved in the following way. The algorithm presented in Sec. 4 translates $\mathcal{S}$ into a TPN (observer) $\mathcal{T}_{\mathcal{S}}$ which recognizes matching traces. $\mathcal{T}_{\mathcal{S}}$ is composed with the SUA $\mathcal{N}$ to check whether a matching execution exists, by using available model checking tools for TPNs. Specifically, the model-checking problem consists in verifying whether there exists an execution that reaches a state where place $\mathsf{validMatch}$ of $\mathcal{T}_{\mathcal{S}}$ is marked, and remains marked thereafter.

Property 6. *Given $\mathcal{N}$ and $\mathcal{S}$ then: $\mathcal{N}\|\mathcal{T}_{\mathcal{S}}$ is bounded iff $\mathcal{N}$ is bounded.*

Property 7. *Given $\mathcal{N}$, $\mathcal{S}$ with $\Sigma_{\mathcal{S}} \subseteq \Sigma_{\mathcal{N}}$, and a trace σ over $\Sigma_{\mathcal{N}} \cup \{\lambda\}$ then: σ is a trace of $\mathcal{N}\|\mathcal{T}_{\mathcal{S}}$ iff σ is trace of $\mathcal{N}$.*

Therefore, we are sure that the composition of $\mathcal{N}$ with the TPN $\mathcal{T}_{\mathcal{S}}$ of the scenario preserves the traces of the SUA.

Theorem 8 (Model checking VTS). *Given $\mathcal{N}$ and $\mathcal{S}$ with $\Sigma_{\mathcal{S}} \subseteq \Sigma_{\mathcal{N}}$, then: $\mathcal{N} \models \mathcal{S}$ iff there exists a time-divergent execution sequence ρ of $\mathcal{N}\|\mathcal{T}_{\mathcal{S}}$ such that, $\exists k \in \mathbb{N}.\ \forall k' \geq k.\ m_{\rho_{k'}}(\mathsf{validMatch}) = 1$.*

6 Case studies

To carry out our tests, we resort to a *tool chain* that allows us to link the VTS scenarios with AADL models. Based on a property expressed as a VTS conditional scenario, we use the tool presented in [11] to generate the related VTS existential scenarios, that are then translated into TPNs. For this last step, we have developed a tool that implements the translation algorithm described in Sec. 4. On the other hand, the TPN representing the AADL models have been constructed manually[9]. Finally we use the composition of both resulting TPNs as input to the tool Tina, which generates the reachability graph preserving LTL. To check whether the model satisfies the property, we encode Thm. 8 as an LTL model-checking problem and use the `selt` application of the Tina tool-box. For the case studies we analyzed, because `selt` is unable to determine whether an execution is time-divergent, we either relied on the *strongly non-Zeno* [15] hypothesis of the SUA or performed semi-automatic verification. We discuss in the conclusions an approach for automatizing the procedure derived from Thm. 8.

6.1 AADL Mode Change Protocol

In AADL systems, components can operate in different modes, where each of them is associated with a configuration of the component. Changes between modes are triggered by events. A more detailed description can be found in [6].

[9] In the future we plan to use OCARINA [10] or TOPCASED (through FIACRE [3]) to generate them automatically.

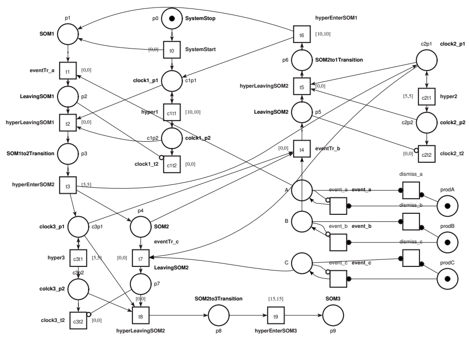
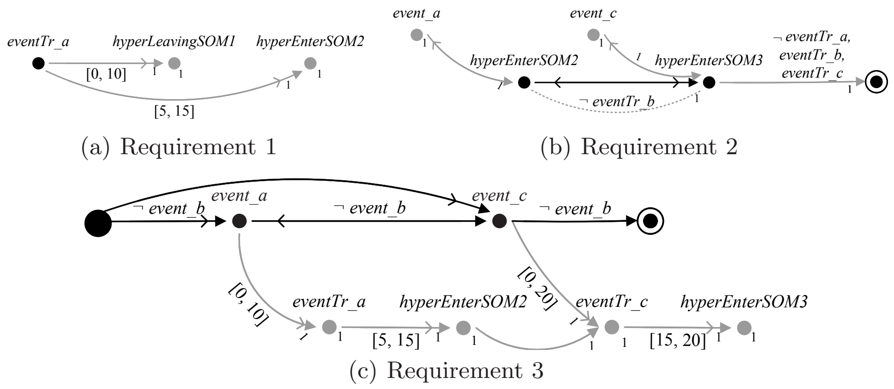


Fig. 14. TPN of the model driver

Fig. 14 shows the TPN of a driver system (extracted from [6]). *VTS* can be used to analyze and verify different kinds of properties. The mode-change protocol should ensure that the maximum delay between a mode-change request and the entry in the new mode is lower than a specified value. Fig. 15(a) shows a conditional scenario for the verification of this property at the request of *event_a*. Fig. 15(b) expresses the correlation between the driver events with the environment ones. For example, part of this conditional scenario establishes that if a change to mode SOM2 occurs, a corresponding input event *event_a* triggering the mode-change must have occurred. Fig. 15(c) presents a conditional scenario where the antecedent defines an environment behavior by which a certain driver property (the consequent) must be verified. It is important to notice that with *VTS* we avoid modelling the environment as a (hand-coded) TPN composed with the driver model as proposed at [6], by including its behavior in the scenario as its antecedent. All these scenarios were verified to hold.



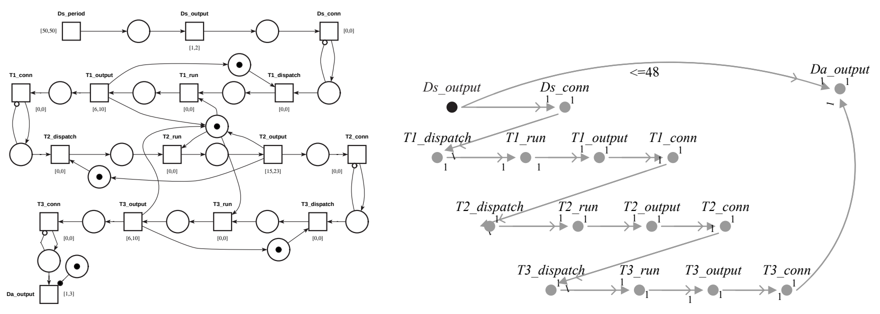
(a) Requirement 1 (b) Requirement 2

(c) Requirement 3

Fig. 15. *VTS* Conditional Scenarios for verifying Mode-Change example

6.2 AADL Flows Specification

AADL flow specifications are used to describe externally observable sequences of connections through component ports. Flow specifications can be annotated with properties, such as latency, whose verification depend on the properties of the involved components, ports, etc., such as execution times, periods, deadlines, communication delays, etc. Quantitative analysis of flow properties is addressed in [8] and implemented in OSATE. The proposed technique, is based on case-by-case analysis according to the architecture of the sub-components. Here, we propose using VTS scenarios for checking flow latency. We believe the advantages of our approach are twofold. First, it is independent of the architecture of the SUA. Second, it allows specifying non-linear flows, currently not available in AADL. As a case study, we use the example provided in [8]. The TPN of the 3-task system with a periodic sensor and aperiodic tasks and actuator is shown in Fig. 16(a). The VTS scenario for the flow specification is shown in Fig. 16(b). This scenario asserts two properties at once: whenever the sensor produces an output, then (1) the flow is realized, and (2) its latency is less than or equal to 48. Notice that our approach gives a tighter latency than the one in [8].



(a) TPN of aperiodic tasks (b) VTS scenario of flow latency

Fig. 16. Flow latency example (taken from [8])

7 Conclusions and Future Works

This paper proposes an approach for checking complex properties on AADL specifications by relying on the visual language VTS for expressing them. To make it practical, we devised a procedure for generating TPNs from VTS to enable its connection with available IDEs for AADL, such as OSATE and TOPCASED, which integrate TPN-based verification tools.

VTS scenarios proved to be adequate to intuitively express complex properties of AADL models. We also incorporate the idea of using them to describe flows in a more general and independent way. Besides its concrete practical application to AADL-centric system design, the translation presented in this work provides an alternative way to verifying VTS requirements in addition to the one based upon timed automata reachability analysis [1].

Several future research directions are envisaged. First, we plan to generate TOPCASED tool-independent intermediate modeling language FIACRE [3] instead of TPN directly. This will allow model-checking VTS with a larger number of tools integrated by the TOPCASED consortium. Second, we will explore more deeply the connection between VTS and AADL flows. The purpose of this is to investigate whether AADL flow specifications could be extended to cope with non-linear flows. Last but not least, to fully automatize the approach resulting from Thm. 8, a verification procedure which takes into account time-divergence should be implemented for TPNs, adapting, for instance, the algorithms proposed in [15] for timed Büchi automata.

References

1. A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual Timed Event Scenarios. In *Proc. of the 26th ACM/IEEE ICSE '04*. ACM Press, 2004.
2. K. Altisen, G. Gossler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. RTSS '99*. IEEE Comp. Soc. Press, 1999.
3. B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat. Fiacre: An intermediate language for model verification in the topcased environment. In *4th European Conf. ERTS*, January 2008.
4. B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In *3rd Int. Conf. on the Quantitative Evaluation of Systems - (QEST'06)*, 2006.
5. B. Berthomieu and F. Vernadat. State Space Abstractions for Time Petri Nets. In *Handbook of Real-Time and Embedded Systems*, Crc Computer & Information Science Series. Chapman & Hall, July 2007.
6. D. Bertrand, A.-M. Déplanche, S. Faucou, and O. H. Roux. A Study of the AADL Mode Change Protocol. In *13th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE, 2008.
7. V. Braberman, N. Kicillof, and A. Olivero. A Scenario-Matching Approach to the Description and Model Checking of Real-Time Properties. *IEEE Transactions on software Engineering*, 31(12), 2005.
8. P. Feiler and J. Hansson. Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). Technical Note CMU/SEI-2007-TN-010, Carnegie Mellon University, June 2007.
9. G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Romeo: A Tool for Analyzing Time Petri Nets. In *Computer Aided Verification*, pages 418–423. Lecture Notes in Computer Science, 2005.
10. J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems*, Oct. 2008.
11. D. Monteverde. Verificación Automática de Escenarios Condicionales. Master's thesis, FCEyN. Univ. de Buenos Aires, 2007.
12. D. Monteverde, A. Olivero, S. Yovine, and V. Braberman. VTS-based specification and verification of behavioral properties of AADL models. Technical report, DC. FCEN. UBA. http://www.dc.uba.ar/people/exclusivos/vbraber, 2008.
13. SAE. Architecture Analysis and Design Language. SAE Standard AS5506, 2004.
14. F. Singhoff, A. Plantec, and P. Dissaux. Can We Increase the Usability of Real Time Scheduling Theory? The Cheddar Project. In *Ada-Europe 2008*, 2008.
15. S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed buchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3), May 2005.