Dynamic Timed Automata for Reconfigurable System Modeling and Verification

Samir Tigane, Fayçal Guerrouf, Nadia Hamani, Laid Kahloul, Mohamed Khalgui, Masood Ashraf Ali

axioms

MDPI

Article

Dynamic Timed Automata for Reconfigurable System Modeling and Verification

Samir Tigane [1]*, Fayçal Guerrouf [1], Nadia Hamani [2], Laid Kahloul [1], Mohamed Khalgui [3] and Masood Ashraf Ali [4]

[1] LINFI Laboratory, Computer Science Department, University of Biskra, Biskra 07000, Algeria
[2] LTI Laboratory, University of Picardie Jules Verne, 02100 Saint-Quentin, France
[3] National Institute of Applied Sciences and Technology, University of Carthage, Tunis 1080, Tunisia
[4] Department of Industrial Engineering, College of Engineering, Prince Sattam bin Abdulaziz University, Al-Kharj 11942, Saudi Arabia
* Correspondence: s.tigane@univ-biskra.dz

Abstract: Modern discrete-event systems (DESs) are often characterized by their dynamic structures enabling highly flexible behaviors that can respond in real time to volatile environments. On the other hand, timed automata (TA) are powerful tools used to design various DESs. However, they lack the ability to naturally describe dynamic-structure reconfigurable systems. Indeed, TA are characterized by their rigid structures, which cannot handle the complexity of dynamic structures. To overcome this limitation, we propose an extension to TA, called dynamic timed automata (DTA), enabling the modeling and verification of reconfigurable systems. Additionally, we present a new algorithm that transforms DTA into semantic-equivalent TA while preserving their behavior. We demonstrate the usefulness and applicability of this new modeling and verification technique using an illustrative example.

Keywords: dynamic timed automata; formal modeling and verification; graph transformation; reconfigurable systems; timed automata; UPPAAL

MSC: 68Q45; 68Q60; 93B17; 68Q85

Citation: Tigane, S.; Guerrouf, F.; Hamani, N.; Kahloul, L.; Khalgui, M.; Ali, M.A. Dynamic Timed Automata for Reconfigurable System Modeling and Verification. *Axioms* **2023**, *12*, 230. https://doi.org/10.3390/axioms12030230

Academic Editors: Xi Deng and Muhammad Azeem Akbar

Received: 1 January 2023
Revised: 18 February 2023
Accepted: 20 February 2023
Published: 22 February 2023

1. Introduction

In recent years, dynamic-structure systems such as cloud computing [1,2], smart grids [3], Industry 4.0 [4,5], and the Internet of things [6] have gained much attention. The design of these highly complex systems is a hot topic, yet it remains a challenging issue that requires suitable and rigorous frameworks [7].

As an intuitive model-checking technique, timed automata [8] (TA) have also found a wide range of applications in the modeling and verification of discrete-event systems (DESs). Several works have used timed automata and their toolbox UPPAAL [9–11] as a modeling and simulation environment [12–16]. However, with the significant development of technologies in recent reconfigurable DESs, state-transition systems such as TA and Petri nets [17] exhibit several shortcomings in the design of these modern systems. They do not enable dynamic structure modeling due to their static structure. To address this issue, many researchers have introduced graph transformation systems (GTSs) into state-transition systems to enable their structures to be dynamic [18–21]. GTSs are well suited for designing complex dynamic systems; however, their high expressiveness impairs any automatic analysis due to their undecidability property. To the best of our knowledge, there is no literature that introduces GTSs into TA to model dynamic structures.

In this paper, we propose a new formalism, called dynamic timed automata (DTA), which allows the modeling and verification of dynamic-structure DESs by handling dynamic typologies in TA. We also provide a new algorithm that transforms DTA into semantically equivalent TA, which unfolds reachable configurations of a given DTA into a

basic TA that preserves the original behavior. This enables a model-checking process to be carried out using analysis methods supported by off-the-shelf tools, such as UPPAAL. It is important to note that the proposed approach focuses on the properties of a reconfigurable system under consideration and does not involve the verification of GTSs themselves. For instance, a critical pair analysis (CPA) [22], used to detect dependencies and conflicts between transformation rules, is not considered since it is beyond the scope of this research. Nevertheless, as suggested in [23], a CPA can assist a designer in detecting dependencies or conflicts by using existing tools (e.g., AGG [24]). The designer can then modify the models or keep the conflicts and dependencies. After resolving these issues, the designer can apply the GTS to a TA modeling an initial configuration to obtain the set of all possible configurations.

The remainder of this paper is organized as follows. An overview of some related works is provided in Section 2. Section 3 recalls the necessary basics of TA and graph transformations. Sections 4 and 5 present the proposed DTA formalism and its unfolding towards TA. The semantic equivalence between any given DTA and its unfolding towards TA is proven in Section 6. Section 7 exploits the proposed formalism to design a reconfigurable system. Finally, Section 8 concludes the paper.

2. Related Work

In the literature, an extensive amount of research has used timed automata to model and verify various kinds of real-time systems [12–16]. The automated consistency checks of the reconfiguration steps of dynamic software product lines (DSPLs) were facilitated in [25] by translating real-time requirements into TA. To improve the performance of consistency checks for DSPL specifications based on TA, different static-analysis techniques were combined in [26]. To guarantee the consistency of resource delivery in a cloud service, the authors of [27] used UPPAAL for the modeling and verification of clients, service managers, and resource services to offer a framework of resource provisioning as a service in the cloud. To handle the dynamics of reputation, the authors in [28] designed a calculus of mobile agents to cope with such features and then encoded them into networks of weighted TA.

Timed automata have been adapted to various formalisms to tackle different challenges. One such adaptation is parametric timed automata (PTA) [29], which allows for more realistic timing constraints. For instance, a constraint that states "an action X must occur within the time it takes to execute n actions Y" is more realistic than a constraint that states "an action X must occur within m milliseconds". This allows the creation of specifications that are based on certain parameters of the environment in which a system operates. Studies in [29,30] found that the problem of determining whether a certain state was reachable was decidable in two scenarios: when using PTA with a single parametric clock, and when using PTA with two parametric clocks and a single parameter. However, the problem of determining reachability was undecidable when using PTA with three or more parametric clocks.

Cordy et al. [31] proposed featured timed automata (FTA) to model and verify the variability in software product lines (SPLs). In FTA, clock constraints on switches and locations were annotated with feature constraints. Hence, instead of considering every product variant one-by-one during the verification process (which is infeasible), entire product lines could be verified in a single run. Although FTA provided a powerful tool, their expressiveness only allowed Boolean feature constraints. To enable unbounded timing intervals of real-time constraints in FTA, Luthmann et al. [32] defined configurable PTA (CoPTA) combining FTA and PTA. However, due to their nature, CoPTA potentially comprised an infinite number of TA model variants. Thus, a variant-by-variant analysis strategy for CoPTA was impossible. To tackle this issue, Luthmann et al. [33] adapted a family-based test-suite generation methodology presented in [34] to CoPTA models.

Latreche and Belala [35] developed a new type of timed automata called recursive and dynamic timed automata (RDTA) which allowed the automata to be recursively invoked by other automata. These were used to analyze and specify procedures for recovering from

failures and updating partner services in dynamic Web service compositions. Another formalism, called dynamic networks of timed automata (DNTA), was introduced in [36] to allow for the creation and destruction of multiple copies of automata at runtime for modeling complex systems. In [37], the authors presented dynamic input/output automata (DIOA) which allowed the dynamic creation, destruction, and changes in the signature of an automaton. The works provided in [38,39] described an extension of TA, called reconfigurable hierarchical TA (RHTA), which involved manually constructing all reachable configurations and incorporating them into a single model which could often lead to cumbersome models. Moreover, a reconfiguration in RHTA was expressed by a simple switching between configurations.

Although the cited extensions are considered practical, provide a certain degree of flexibility in the modeling, and enhance the expressiveness of TA, they cannot model or verify dynamic-structure automata. For instance, FTA and CoPTA can model variability in SPLs; however, each variant is static and cannot change its structure. As for RDTA, they can only specify and analyze the backward recovery procedures and the update of partner services in case of failure; thus, neither new structures nor other behaviors are allowed. Even though DNTA and DIOA enable dynamic sets of automata by instantiating or destroying them, the structure of any automaton remains static. Furthermore, DIOA are based on untimed automata, and they can only exhibit or hide certain behaviors without allowing the introduction of new ones. Finally, RHTA represent reconfigurations by static models, which can only make the modeling process laborious at best, therefore losing the benefits of a direct representation [37].

The formalism proposed in this paper outperforms the above-mentioned extensions in several ways:

- It enables the separation of concerns by modeling dynamic structures using GTSs, which explicitly model system features and component set evolution at two separate levels [40];
- It provides an unfolding algorithm that transforms DTA to semantic-equivalent TA, thereby making the design and verification of dynamic systems more efficient by reusing the existing TA tools in the analysis of DTA;
- All properties that are decidable in the TA formalism remain decidable in the new extension, since any given DTA can be unfolded to a plain TA that preserves the behavior of its original DTA.

3. Preliminaries

This section outlines the necessary basics of the formalisms exploited in this work.

3.1. Timed Automata

Timed automata (TA) [8] extend finite automata by introducing real-valued *clocks* to accept timed languages. Let C be a set of clocks. For each clock $x \in C$, the following is considered:

- Initially, the value of x is zero;
- Its value increases simultaneously with other clocks by the same speed;
- It can be reset to zero with any edge.

Let $\mathcal{G}(C)$ be the set of conjunctive formulae of atomic constraints built over C of the form $x \bowtie a$ or $x - y \bowtie a$, where $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and $a \in \mathbb{N}$. The formal definition of a timed automaton is provided as follows.

Definition 1. ***(Timed automaton).*** *A TA A is a tuple $\langle S, s_0, \Sigma, C, E, I \rangle$ where:*

(1) *S is a nonempty finite set of locations;*
(2) *$s_0 \subset S$ is a set of initial locations;*
(3) *Σ is a finite set of actions containing an internal action denoted by τ;*
(4) *C is a finite set of clocks;*

(5) $E \subseteq S \times \Sigma \times \mathcal{G}(C) \times 2^C \times S$ *is a set of edges between locations, where* $(s, \sigma, g_c, \delta, s') \in E$ *means the following;*
   (a) *s and s' are the source and the target locations, respectively;*
   (b) *σ is an action;*
   (c) *g_c is an enabling condition built over C;*
   (d) *$\delta \subseteq C$ is a subset of clocks to be reset.*

(6) $I : S \to \mathcal{G}(C)$ *assigns invariants to locations.*

Example 1. *Consider a TA A depicted in Figure 1 that models a system with three states:* `working`, `repairing`, *and* `fail_safe`. *Initially, A is in its initial location* `working` *depicted by a double circle. When a problem arises, the automaton changes its location to* `repairing` *and resets clock* `x` *(i.e.,* `x:=0`*). At location* `repairing`*, the system should be repaired within 15 time units. If the system is repaired before the deadline, it returns to its initial state* `working`*. If not, it changes its state to* `fail_safe` *mode and resets clock* `x`*. Finally, after entering* `fail_safe`*, any breakdown must be repaired before 50 time units have passed, after which A returns to location* `working`*.*

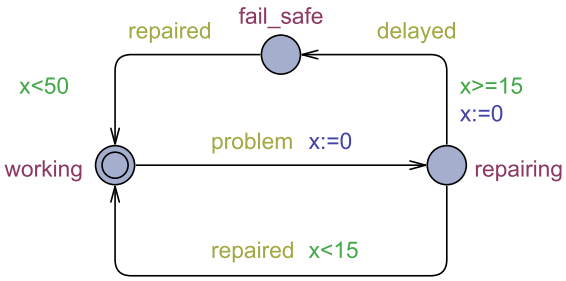


Figure 1. A TA modeling a system having three states.

UPPAAL increases the expressiveness of TA by introducing constants, discrete local/global variables, and the communication channels [11] used by automata to communicate with each other. Indeed, a set of automata communicating on (shared) global variables and channels constitutes a global context called a network of TA (NTA).

For an NTA composed of a set of automata, a set of synchronized channels `Chan`, and a set of global variables $\mathtt{V}^{\mathtt{g}}$, consider the following.

- Let $\mathtt{Sync} = \{\mathtt{c!}, \mathtt{c?} \mid \mathtt{c} \in \mathtt{Chan}\} \cup \{-\}$ be a set of synchronizations, such that `c!` and `c?` represent the initiation and the acceptance, respectively, of synchronization over channel `c`, and "$-$" stands for no synchronization.
- Let $\mathtt{V} = \mathtt{V}^{\mathtt{g}} \cup \mathtt{V}^{\mathtt{l}}$ be a set of variables.
- Let $\mathtt{Guards}(\mathtt{V})$ be the set of logical conditions built over $\mathtt{V}$.
- Let $\mathtt{Exp}(\mathtt{V})$ be the set of expressions built over $\mathtt{V}$.
- Let $\mathtt{Assign}(\mathtt{V})$ be the set of finite sequences of assignments of the form `v:=exp`, where $\mathtt{v} \in \mathtt{V}$ and $\mathtt{exp} \in \mathtt{Exp}(\mathtt{V})$.

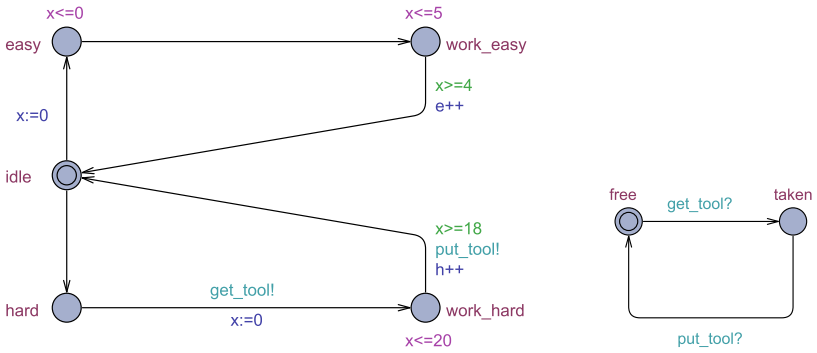
Definition 2. ***(Timed automaton in UPPAAL).*** *A TA in UPPAAL is a tuple* $\langle \mathtt{V}^{\mathtt{l}}, S, s_0, \Sigma, C, E, I \rangle$ *such that:*

(1) $\mathtt{V}^{\mathtt{l}}$ *is a set of initialized local variables (a variable in UPPAAL can be a real or an integer);*

(2) *S, s_0, Σ, C, and I are as in Definition 1;*

(3) $E \subseteq S \times \Sigma \times \mathcal{G}(C) \times \mathtt{Guards}(\mathtt{V}) \times \mathtt{Sync} \times 2^C \times \mathtt{Assign}(\mathtt{V}) \times S$ *is a set of edges, where for* $e = (s, \sigma, g_c, g_v, z, \delta, \alpha, s') \in E$*:*
   (a) *s, σ, g_c, δ, and s' are as provided in Definition 1;*
   (b) *g_v is an enabling condition built over* $\mathtt{V}$*;*
   (c) *z is a synchronization on a channel* `c`*. Note that the synchronization can take place only if an edge e of automaton A is sending on* `c` *(i.e.,* `c!`*) and an edge e' of automaton A' is receiving on* `c` *(i.e.,* `c?`*);*
   (d) *α is a sequence of assignments updating the values of variables in* $\mathtt{V}$*.*

Example 2. *Consider a network of timed automata shown in Figure 2. This NTA models a job shop (inspired by [10]) consisting of workers (see Figure 2a) and tools (see Figure 2b). In this job shop, workers share certain tools to manufacture products from simple components. A worker picks up some components and decides whether to perform an easy task with their hands, i.e., no tool is involved, or a hard task using an available tool.*



(**a**) A worker (**b**) A tool

Figure 2. A network of timed automata modeling a job shop.

Initially, each worker is in location `idle` *and each tool is in location* `free`*. If a worker decides to perform an easy task, they start working on it straightaway. This behavior is modeled by moving from location* `idle` *to location* `easy` *and then to location* `work_easy`*. The invariant "*`x<=0`*" associated with location* `easy` *models that a worker needs no time to start an easy task. In addition, the invariant "*`x<=5`*" associated with location* `work_easy` *means that an easy task is performed at most in five time units. After finishing an easy task, a global variable called "*`e`*" (the number of finished easy tasks) is incremented.*

As for hard tasks, moving from location `hard` *to location* `work_hard` *requires the existence of an available tool. This is modeled by both synchronizations "*`get_tool!`*" and "*`get_tool?`*" shown in Figure 2a,b, respectively. That is, a worker automaton sending on channel "*`get_tool`*" can move from location* `hard` *to* `work_hard` *iff there is a tool automaton receiving on "*`get_tool`*". Finally, after finishing a hard task, a global variable called "*`h`*" (the number of finished hard tasks) is incremented and the synchronization on channel "*`put_tool`*" is initiated.*

3.2. Graph Transformation: A Double-Pushout Approach

In this work, the reconfiguration of timed automata is modeled based on a widely used graph transformation formalism called double-pushout (DPO) approach. This approach provides a theoretical and suitable framework for modeling dynamic-structure systems in a rigorous way [18].

We start by defining graph morphism, an important concept in graph transformation. It preserves the structure of a graph G in a graph H by mapping nodes of G to nodes of H so that each edge in G must have an image in H.

Definition 3. ***(Graph morphism).*** *Let V_G and E_G denote sets of nodes and edges of a graph G, respectively. Let $G = \langle V_G, E_G \rangle$ and $H = \langle V_H, E_H \rangle$ be two graphs. A function $\varphi : V_G \to V_H$ is a graph morphism if the following holds: $\forall (v, w) \in E_G, (\varphi(v), \varphi(w)) \in E_H$.*

In DPO, a transformation is provided as a rule $r = L \xleftarrow{\varphi_l} I \xrightarrow{\varphi_r} R$ consisting of three graphs L, I, and R and two graph morphisms φ_l and φ_r, where:

- L is a left-hand side (to be removed);
- I is a common interface;
- R is a right-hand side (to be inserted);
- $\varphi_l : I \to L$ is a graph morphism;
- $\varphi_r : I \to R$ is a graph morphism.

Given a rule r, elements in L that do not belong to the image of φ_l are called *obsolete* and elements in R that do not belong to the image of φ_r are called *fresh*. A rule r applies to a graph G if an occurrence (located by a graph morphism m) of its left-hand side L exists in G. If graph morphisms m, φ_l, and φ_r are injective, then the obsolete elements of L are removed from G (the obtained graph is called *context*) and fresh elements of R are added to the context graph, resulting in a new graph H. This transformation is denoted by $G \overset{r,m}{\Longrightarrow} H$. In this paper, we only considered injective graph morphisms. The DPO-based transformations are also represented by diagrams, as depicted in Figure 3.

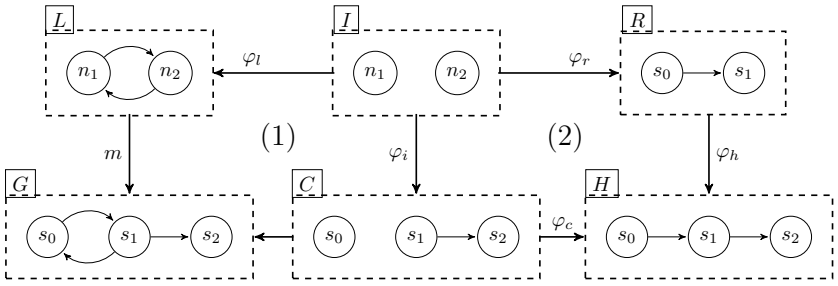


Figure 3. A DPO diagram.

A DPO transformation performs two steps called *pushout*. Informally, the first pushout (see Box (1) in Figure 3) resolves the existence of a *context* graph C to be glued (the graph gluing is explained below) with L over I to obtain G, denoted by $G = C +_I L$. If C exists, then the rule applies to G. The second pushout (see Box (2) in Figure 3) glues C with R over I to yield graph H.

The obtained graph $H = \langle V_H, E_H \rangle = C +_I R$ is defined as follows.

1. $V_H = V_C \uplus (V_R \setminus V_I)$ (such that "$\uplus$" denotes the disjoint union);
2. $E_H = E_C \uplus E_R$.

Consider an application of a rule $r = L \overset{\varphi_l}{\longleftarrow} I \overset{\varphi_r}{\longrightarrow} R$ to a graph G shown in Figure 3. The first pushout is performed as follows. A morphism m locates an occurrence of L in G, such that $m(n_1) = s_0$ and $m(n_2) = s_1$. Then, a context graph C is defined so that $G = C +_I L$. Hence, the rule is applicable. In the second pushout, nodes and edges of C and a fresh edge of R (which is (s_0, s_1)) are inserted into H.

4. Dynamic Timed Automata

In this paper, we present a new formalism, called dynamic timed automata (DTA), that overcomes the limitations of traditional TA in modeling dynamic-structure reconfigurable systems. We also present a new algorithm that transforms DTA into equivalent TA while preserving their behavior.

This section presents the formal definition of the DPO-based DTA formalism. In the following sections, we develop an algorithm for unfolding DTA into semantic-equivalent TA, and we prove the equivalence between DTA and their unfolding towards TA through the proposed algorithm.

To begin, we extend the definition of graph morphism provided in Definition 3.

Definition 4. ***(TA morphism).*** *Let* $A_1 = \langle V_1^l, S_1, s_{01}, \Sigma_1, C_1, E_1, I_1 \rangle$ *and* $A_2 = \langle V_2^l, S_2, s_{02}, \Sigma_2, C_2, E_2, I_2 \rangle$ *be two TA, where* $V_1^l \subset V_2^l$, $\Sigma_1 \subset \Sigma_2$, *and* $C_1 \subset C_2$. *A TA morphism* $\varphi : A_1 \to A_2$ *is a mapping* $\varphi : S_1 \to S_2$ *such that the following hold.*

- $\forall\, s \in S_1$, *if* $s \in s_{01}$, *then* $\varphi(s) \in s_{02}$. *(Note that* s_{01} *might be empty);*
- $\forall\, e_1 = (s_1, \sigma, g_c, g_v, z, \delta, \alpha, s_1') \in E_1, \exists\, e_2 = (s_2, \sigma, g_c, g_v, z, \delta, \alpha, s_2') \in E_2, \varphi(s_1) = s_2$ *and* $\varphi(s_1') = s_2'$;
- $\forall\, s \in S_1, I_1(s) = I_2(\varphi(s))$.

Definition 5. ***(Transformation rules).*** *A transformation rule r is defined as $\langle L \xleftarrow{\varphi_l} I \xrightarrow{\varphi_r} R, g, u \rangle$, such that:*

1. *L is a left-hand side TA;*
2. *I is a common interface TA;*
3. *R is a right-hand side TA, $|S_L| = |S_I| = |S_R|$, such that S_A denotes the location set of A;*
4. *$\varphi_l : I \to L$ is a TA morphism;*
5. *$\varphi_r : I \to R$ is a TA morphism;*
6. *$g = (s, g_c, g_v)$ is a precondition of r such that s is a location and g_c (resp. g_v) is an enabling condition built over clocks (respectively, variables);*
7. *$u = (\delta, \alpha)$ is a post-condition (i.e., effect) of r such that δ is a set of clocks to be reset and α is a sequence of assignments.*

Moreover, rules that can only modify the set of edges (i.e., topology) of an automaton are considered in this work.

Definition 6. ***(Dynamic timed automata).*** *A DTA $\mathcal{D}$ is a pair $\langle A_0, \mathcal{R} \rangle$, such that:*

1. *A_0 is a TA;*
2. *$\mathcal{R}$ is a set of transformation rules.*

Let A be a TA. A rule $r = \langle L \xleftarrow{\varphi_l} I \xrightarrow{\varphi_r} R, g, u \rangle$ with a match $m : L \to A$ applies to a DTA $\mathcal{D}$ iff:

1. A is the current configuration of $\mathcal{D}$;
2. There exists a TA C such that $A = C +_I L$;
3. $g = (s, g_c, g_v)$ is satisfied, that is, s is the current location of A, and the valuations of both guards g_c and g_v are true.

After applying $r = \langle L \xleftarrow{\varphi_l} I \xrightarrow{\varphi_r} R, g, u \rangle$ to A, $\mathcal{D}$ changes its configuration towards A' such that $A' = C +_I R$.

Example 3. *Consider a DTA $\mathcal{D} = \langle A, \{r\} \rangle$, where TA A and rule r are depicted in Figure 4. Rule $r = \langle L \xleftarrow{\varphi_l} I \xrightarrow{\varphi_r} R, g, u \rangle$ is defined as follows:*

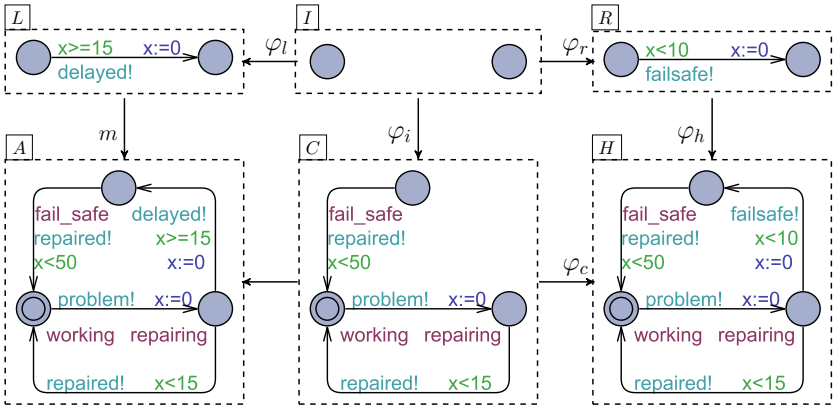


Figure 4. Reconfiguration via rule r.

1. *L, I, R, φ_l and φ_r are shown in Figure 4;*
2. *$g = (s, g_c, g_v)$, such that $s =$ `repairing`, $g_c =$ `"x<20"` and $g_v =$ `"true"`;*
3. *$u = (\delta, \alpha)$, such that $\delta = \{$`x`$\}$ and $\alpha = \varepsilon$ (i.e., an empty sequence).*

Rule r applies to A, given the following:

- *Its precondition g is satisfied (i.e., the current location of A is* `repairing` *and the value of clock* `x` *is less than 20);*
- *A morphism m finds an occurrence of L in A, such that the left and right locations of L are mapped to* `repairing` *and* `fail_safe`*, respectively;*

- *There exists a TA C, shown in Figure 4, such that $A = C +_I L$.*
  After applying r, the current configuration of $\mathcal{D}$ becomes H illustrated in Figure 4.

5. DTA Transformation towards Basic TA

Timed automata enjoy a variety of tools and frameworks exploited in their design and analysis. This fact motivates the unfolding of DTA into TA. Therefore, the traditional verification tools designed for basic TA can be utilized in the DTA analysis.

In fact, reconfigurations insert and/or remove edges to enable new behaviors/topologies. As a result, any given unfolding of a DTA must model these reconfigurations and their effects. In this context, we developed an algorithm that unfolds any given DTA $\mathcal{D}$ into a semantic-equivalent TA $\mathcal{A}$ according to its shape and behavior.

For a DTA $\mathcal{D} = \langle A_0, \mathcal{R} \rangle$, consider the following:

1. Let $\mathcal{C} = \{A_0, \ldots, A_n\}$ be a set of TA obtained by applying sequences of rules in $\mathcal{R}$ to A_0.
2. Let E_i be the set of edges in A_i.
3. Let $\mathcal{E} = \bigcup_{i=0}^{n} E_i$ be a set of edges.
4. Let $\mathcal{C}(e) = \{A_i \mid A_i \in \mathcal{C} \text{ and } e \in E_i\}$.
5. Let $\mathcal{T} = \{(A^s, r, A^t) \mid A^s, A^t \in \mathcal{C} \text{ and } A^s \overset{r}{\Rightarrow} A^t\}$ be a set of transformations applicable to DTA $\mathcal{D}$, where r is an applicable rule to $\mathcal{D}$, and A^s, A^t are its source and target configurations, respectively.

The intuition of the proposed unfolding algorithm provided in Algorithm 1 that transforms a given DTA $\mathcal{D}$ to a TA $\mathcal{A}$ is given as follows:

1. Let $A_0 = \langle \mathtt{V}_0^{\mathtt{l}}, S_0, s_{00}, \Sigma_0, C_0, E_0, I_0 \rangle$ be an initial configuration of $\mathcal{D}$.
2. Let $\mathtt{Enum} : \mathcal{C} \longrightarrow \{0, \ldots, n\}$ be an enumeration of configurations $A_0, \ldots, A_n$ in $\mathcal{C}$ by means of a natural ordering.
3. Let $\mathtt{V}^{\mathtt{l}} \leftarrow \mathtt{V}_0^{\mathtt{l}} \cup \{\mathtt{cfg}\}$, where `cfg` is a bounded local integer variable (initialized to zero) used to represent the current configuration of $\mathcal{D}$, that is, if `cfg=i` then the current configuration of $\mathcal{D}$ is A_i, where $\mathtt{Enum}(A_i) = \mathtt{i}$.
4. Let $E \longleftarrow \emptyset$.
5. For each edge $e \in \mathcal{E}$ that is present in every configuration in $\mathcal{C}$, i.e., $\mathcal{C}(e) = \mathcal{C}$, insert e into E, i.e., $E \longleftarrow E \cup \{e\}$.
6. For each edge $e = (s, \sigma, g_c, g_v, z, \delta, \alpha, s') \in \mathcal{E}$ that does not belong to certain configurations in $\mathcal{C}$, i.e., $\mathcal{C}(e) \subsetneq \mathcal{C}$, do:
   (a) Build a condition `le` of the form "`cfg==i`$_0$`||...||cfg==i`$_{|\mathcal{C}(e)|}$" (`||` and `&&` stand for "logical or" and "logical and", respectively), where $\mathtt{i}_0, \ldots, \mathtt{i}_{|\mathcal{C}(e)|}$ are the indices, obtained by `Enum`, of configurations in $\mathcal{C}(e)$.
   (b) Create $e' = (s, \sigma, g_c, g'_v, z, \delta, \alpha, s')$, where $g'_v =$"`le && `g_v" (6a).
   (c) Insert e' into E, i.e., $E \longleftarrow E \cup \{e'\}$.
7. For each transformation $(A^s, r, A^t) \in \mathcal{T}$, do:
   (a) Let $g = (s, g_c, g_v)$ and $u = (\delta, \alpha)$ be the pre- and postconditions of rule r.
   (b) Let $g'_v =$"`cfg==i && `g_v", where $\mathtt{i} = \mathtt{Enum}(A^s)$.
   (c) Let $\alpha' =$"α`,cfg:=j`", where $\mathtt{j} = \mathtt{Enum}(A^t)$.
   (d) Create an edge $e = (s, \tau, g_c, g'_v, -, \delta, \alpha', s)$ (recall that τ and "$-$" stand for internal action and no synchronization, respectively).
   (e) Insert e into E, i.e., $E \longleftarrow E \cup \{e\}$.
8. Let $S \leftarrow S_0$, $s_0 \leftarrow s_{00}$, $\Sigma \leftarrow \Sigma_0$, $C \leftarrow C_0$, and $I \leftarrow I_0$.
9. Let $\mathcal{A} = \langle \mathtt{V}^{\mathtt{l}}, S, s_0, \Sigma, C, E, I \rangle$.

Consider Steps (6) and (7). In fact, the former represents an edge e of $\mathcal{D}$ by an edge e' in $\mathcal{A}$, where the condition "`le&&`g_v" enables e' only if (i) the current configuration of $\mathcal{A}$ is one of the configurations in which e appears, and (ii) g_v, the enabling condition of e, is true. By the latter, the algorithm creates an edge e to represent an application of a rule r to source configuration A^s yielding a target configuration A^t, such that: (i) the guards of e are

identical to those of r and, in addition, the current configuration is A^s which is expressed by "`cfg==i`" where `cfg` is a variable used to represent the current configuration of $\mathcal{D}$ and `i` is the index of configuration A^s; and (ii) the effects (clocks to be reset and variables assignments) of e are identical to those of r, and, in addition, the current configuration becomes A^t which is expressed by "`cfg:=j`", where `j` is the index of configuration A^t.

Algorithm 1 DTA transformation towards TA.

Require: $\mathcal{D} = \langle A_0, \mathcal{R} \rangle$: DTA
Ensure: $\mathcal{A} = \langle \mathtt{V^l}, S, s_0, \Sigma, C, E, I \rangle$: TA
1: ***Initialization***:
2: Let $A_0 = \langle \mathtt{V_0^l}, S_0, s_{00}, \Sigma_0, C_0, E_0, I_0 \rangle$
3: $\mathtt{cfg} \longleftarrow 0$
4: $\mathtt{V^l} \longleftarrow \mathtt{V_0^l} \cup \{\mathtt{cfg}\}$
5: $\langle S, s_0, \Sigma, C, I \rangle \longleftarrow \langle S_0, s_{00}, \Sigma_0, C_0, I_0 \rangle$
6: $E \longleftarrow \emptyset$
7: Apply rules in $\mathcal{R}$ to A_0
8: Compute $\mathcal{T}$
9: Compute $\mathcal{C}$
10: Let $\mathtt{Enum} : \mathcal{C} \longrightarrow \{0, \ldots, n\}$ be an enumeration
11: $\mathcal{E} \longleftarrow \emptyset$
12: **for each** $A_i \in \mathcal{C}$ **do**
13: $\quad \mathcal{E} \longleftarrow \mathcal{E} \cup E_i$
14: **end for**
15: ***Represent edges of*** $\mathcal{D}$ ***in*** $\mathcal{A}$:
16: **for each** $e = (s, \sigma, g_c, g_v, z, \delta, \alpha, s') \in \mathcal{E}$ **do**
17: $\quad$ **if** $\mathcal{C}(e) = \mathcal{C}$ **then**
18: $\quad\quad E \longleftarrow E \cup \{e\}$
19: $\quad$ **else**
20: $\quad\quad$ Pick arbitrarily A from $\mathcal{C}(e)$
21: $\quad\quad \mathtt{le} \longleftarrow$ `"cfg=="+Enum`(A)
22: $\quad\quad$ **for each** $A_i \in \mathcal{C}(e) \setminus \{A\}$ **do**
23: $\quad\quad\quad \mathtt{le} \longleftarrow \mathtt{le}$`+"||cfg=="+Enum`$(A_i)$
24: $\quad\quad$ **end for**
25: $\quad\quad e' \longleftarrow (s, \sigma, g_c, \mathtt{le\&\&}g_v, z, \delta, \alpha, s')$
26: $\quad\quad E \longleftarrow E \cup \{e'\}$
27: $\quad$ **end if**
28: **end for**
29: ***Represent rules of*** $\mathcal{R}$ ***in*** $\mathcal{A}$:
30: **for each** $(A^s, r, A^t) \in \mathcal{T}$ **do**
31: $\quad g_v' \longleftarrow$ `"cfg=="+Enum`(A^s)`+"&&`g_v`"`
32: $\quad \alpha' \longleftarrow \alpha$`+",cfg:="+Enum`$(A^t)$
33: $\quad e \longleftarrow (s, \tau, g_c, g_v', -, \delta, \alpha', s)$
34: $\quad E \longleftarrow E \cup \{e\}$
35: **end for**
36: ***Termination***:
37: $\mathcal{A} \longleftarrow \langle \mathtt{V^l}, L, l_0, \Sigma, C, E, I \rangle$

Example 4. *In this example, we applied the unfolding algorithm to DTA $\mathcal{D}$ shown in Figure 4. The resulting TA $\mathcal{A}$ is depicted in Figure 5.*
In the following, we use the proposed algorithm to create an equivalent TA $\mathcal{A} = \langle \mathtt{V^l}, S, s_0, \Sigma, C, E, I \rangle$ to $\mathcal{D} = \langle A, \mathcal{R} \rangle$.

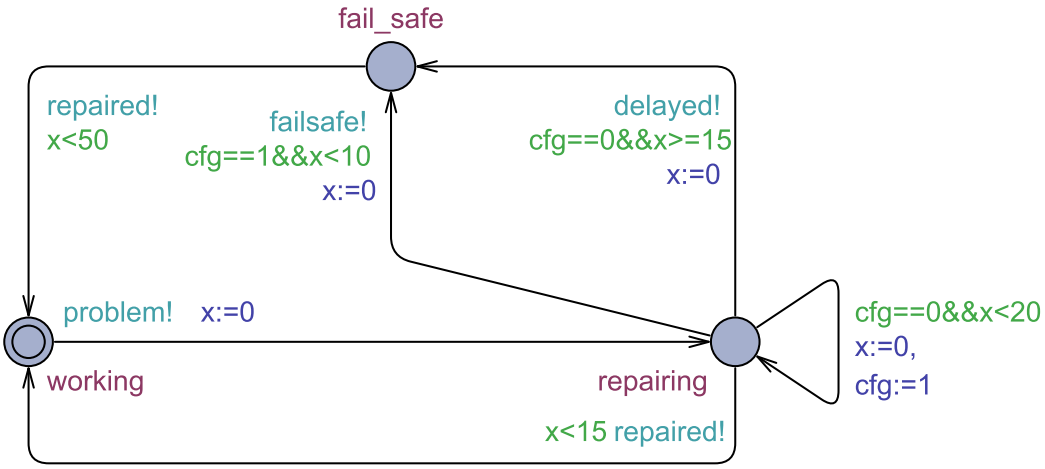


Figure 5. Equivalent TA $\mathcal{A}$ to DTA $\mathcal{D}$.

Initialization: *First, we start the creation of automaton $\mathcal{A}$ such that its sets of locations, initial locations, actions and clocks, and function I are identical to those in A. Additionally, we add a local variable, called* `cfg`*, initialized to zero, to those of A to create a local variable set of $\mathcal{A}$. Thus,*

- $V^l = \{\texttt{cfg}\}$, *where* $V_0^l = \emptyset$;
- $S = \{\texttt{working}, \texttt{repairing}, \texttt{fail\_safe}\}$;
- $s_0 = \{\texttt{working}\}$;
- $\Sigma = \{\tau\}$;
- $C = \{\texttt{x}\}$;
- $I : S \longrightarrow \{\textbf{true}\}$, *where* ***true*** *means there are no invariants on location.*

Second, we apply rules in $\mathcal{R}$ to A to yield two sets: (i) $\mathcal{C} = \{A, H\}$, where configurations A and H are illustrated in Figure 4, and (ii) $\mathcal{T} = \{(A, r, H)\}$. Finally, we enumerate the configurations in $\mathcal{C}$ using a natural ordering; hence, $\texttt{Enum}(A) = 0$ *and* $\texttt{Enum}(H) = 1$.

Represent edges of $\mathcal{D}$ in $\mathcal{A}$: *First, we consider edges in $\mathcal{D}$ that remain unchanged during reconfigurations, i.e., they exist in configurations of $\mathcal{D}$. These edges are:*

- $e_1 = (\texttt{working}, \tau, \textbf{true}, \textbf{true}, \texttt{problem!}, \{x\}, \varepsilon, \texttt{repairing})$;
- $e_2 = (\texttt{repairing}, \tau, \texttt{"x<15"}, \textbf{true}, \texttt{repaired!}, \emptyset, \varepsilon, \texttt{working})$;
- $e_3 = (\texttt{fail\_safe}, \tau, \texttt{"x<50"}, \textbf{true}, \texttt{repaired!}, \emptyset, \varepsilon, \texttt{working})$,

  where ***true*** *indicates "no guards", and ε denotes "an empty sequences".*

  Consider edges that do not appear in every configuration of $\mathcal{D}$, namely:

- $e_A = (\texttt{repairing}, \tau, \texttt{"x>=15"}, \textbf{true}, \texttt{delayed!}, \{x\}, \varepsilon, \texttt{fail\_safe})$ *in A;*
- $e_H = (\texttt{repairing}, \tau, \texttt{"x<10"}, \textbf{true}, \texttt{failsafe!}, \{x\}, \varepsilon, \texttt{fail\_safe})$ *in H.*

Since edge e_A only belongs to configuration A, i.e., $\mathcal{C}(e_A) = \{A\}$, the enabling condition (built over variables) of edge e'_A, which represents e_A in $\mathcal{A}$, is computed as follows "`cfg==0`*&&* ***true****", where* ***true*** *is the enabling condition of e_A and* `0` *is the index of configuration A. Thus, $e'_A = (\texttt{repairing}, \tau, \texttt{"x>=15"}, \texttt{"cfg==0"}, \texttt{delayed!}, \{x\}, \varepsilon, \texttt{fail\_safe})$. Similarly, e'_H, which represents e_H, is defined as follows: $e'_H = (\texttt{repairing}, \tau, \texttt{"x<10"}, \texttt{"cfg==1"}, \texttt{failsafe!}, \{x\}, \varepsilon, \texttt{fail\_safe})$. Finally, edges e_1, e_2, e_3, e'_A, and e'_H are added to set E.*

Adding edges to emulate rules: *An application of r to A results in H, where e_A is no longer present and e_H is newly added. In $\mathcal{D}$, r applies under the following conditions: (i) its current configuration is A, (ii) the value of clock* `x` *is less than 20, and (iii) the current location of A is* `repairing`*. Hence, the preconditions of edge e_r, which represent this application, are (i) "*`cfg==0`*" and (ii) "*`x<20`*"; furthermore, (iii) the source and target location of e_r is* `repairing`*. Moreover, when r applies, (a) clock* `x` *is reset, and (b) the current configuration of $\mathcal{D}$ is changed to H. Therefore, the postconditions of edge e_r are (a) "*`x:=0`*" and (b) "*`cfg:=1`*". Finally, we insert*

$e_r = (\texttt{repairing}, \tau, \texttt{"x<20"}, \texttt{"cfg==0"}, -, \{x\}, \texttt{"cfg:=1"}, \texttt{repairing})$ *into* E. *Recall that* τ *and "*$-$*" stand for internal action and no synchronization, respectively.*

Termination: *Finally, the equivalent TA* $\mathcal{A} = \langle V^l, S, s_0, \Sigma, C, E, I \rangle$ *to DTA* $\mathcal{D}$ *is given as follows.*

- $V^l = \{\texttt{cfg}\}$;
- $S = \{\texttt{working}, \texttt{repairing}, \texttt{fail\_safe}\}$;
- $s_0 = \{\texttt{working}\}$;
- $\Sigma = \{\tau\}$;
- $C = \{x\}$;
- $C = \{e_1, e_2, e_3, e'_A, e'_H, e_r\}$;
- $I : S \longrightarrow \{\textbf{\textit{true}}\}$.

6. Proofs of Termination and Equivalence

This section is dedicated to proving the following three claims: (i) the termination of graph transformations applied to a given DTA $\mathcal{D}$, meaning that its structure is finite; (ii) the termination of the unfolding algorithm; and (iii) the equivalence between a given DTA $\mathcal{D}$ and the TA $\mathcal{A}$ obtained by the unfolding process.

6.1. Graph Transformation Termination

It is important to note that this paper only considers rules that can modify the topology of an automaton, i.e., adding, removing, or modifying edges. Hence, the set of locations in any given DTA is static and finite. Regarding the set of edges after a rule application, we can distinguish three types of rules:

1. Rules that decrease the number of edges, i.e., if $G \overset{r}{\Rightarrow} H$, then $|E_G| > |E_H|$;
2. Rules that preserve the number of edges, i.e., if $G \overset{r}{\Rightarrow} H$, then $|E_G| = |E_H|$;
3. Rules that increase the number of edges, i.e., if $G \overset{r}{\Rightarrow} H$, then $|E_G| < |E_H|$.

Obviously, applying rules of the first and second types will not result in infinite structures, i.e., an infinite set of edges.

In the following, we focus on rules belonging to the third type. Let r be a rule. Applying r to a configuration G requires a morphism m that matches its left-hand side L to a part of G. Since G has a finite structure, the number of occurrences of L in G is also finite. Indeed, if p and n are the number of locations in L and G, respectively, then there are at most $A_n^p = \frac{n!}{(n-p)!}$ occurrences of L in G. Moreover, if $G \overset{r,m}{\Longrightarrow} H_1 \overset{r,m}{\Longrightarrow} H_2$, i.e., r is consecutively applied twice to G with the same morphism m, then $H_1 = H_2$, since adding an existing edge to an automaton does not change its semantic. Accordingly, any consecutive application of r to a configuration G yields at most A_n^p distinct configurations. Finally, if two rules r_1 and r_2 are applied in the order $G \overset{r_1,m_1}{\Longrightarrow} H_1 \overset{r_2,m_2}{\Longrightarrow} H_2 \overset{r_1,m_1}{\Longrightarrow} H_3$, then $H_2 \overset{r_1,m_1}{\Longrightarrow} H_3$ inserts edges already inserted by $G \overset{r_1,m_1}{\Longrightarrow} H_1$ (recall that inserting an existing edge to an automaton does not change its semantic), i.e., $H_2 = H_3$. Therefore, applying any sequence of rules in $\mathcal{R}$ to an initial configuration of DTA $\mathcal{D}$ yields a finite set of configurations $\mathcal{C}$. Accordingly, the set of possible transformations $\mathcal{T} = \{(A^s, r, A^t) | A^s, A^t \in \mathcal{C} \text{ and } A^s \overset{r}{\Rightarrow} A^t\}$ is also finite. Therefore, any graph transformation applied to any given DTA $\mathcal{D}$ terminates.

6.2. Unfolding Termination

The unfolding algorithm described in Section 5 preserves certain finite sets of an initial configuration $A_0 = \langle \mathtt{V}_0^1, S_0, s_{00}, \Sigma_0, C_0, E_0, I_0 \rangle$ of a given DTA $\mathcal{D}$ in its unfolding $\mathcal{A} = \langle \mathtt{V}^1, S, s_0, \Sigma, C, E, I \rangle$, such that $\mathtt{V}^1 = \mathtt{V}_0^1 \cup \{\mathtt{cfg}\}$, $S = S_0, s_0 = s_{00}, \Sigma = \Sigma_0$ and $C = C_0$. As for edges, let $E = E_e \cup E_t$, such that E_e and E_t correspond to edges and transformations in $\mathcal{D}$, respectively.

As proven in the previous subsection, the set of configurations $\mathcal{C} = \{A_0, \ldots, A_n\}$ is finite, and so is the set of edges $\mathcal{E} = \cup_{i=0}^{n} E_i$ belonging to these configurations. On the other

hand, for each edge $e \in \mathcal{E}$ the unfolding algorithm inserts a single edge into the initially empty set E_e to represent e. This means that set E_e is also finite. Similarly, for each possible rule application $G \overset{r}{\Rightarrow} H$, the proposed algorithm inserts a single edge into set E_t. As the set of possible transformations $\mathcal{T}$ is finite, it follows that set E_t is finite as well. Finally, since all sets involved in the computation of an unfolding $\mathcal{A}$ of a given DTA $\mathcal{D}$ are finite, the unfolding algorithm terminates.

6.3. Equivalence between DTA and Their Unfolding TA

In order for a given DTA to be equivalent to its unfolding TA, the latter must preserve the behavior of the former. That is, if a run $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \dots s_q \xrightarrow{r} s_q \dots s_{n-1} \xrightarrow{e_n} s_n \dots$, where e_i is an edge and r is a rule, is recognized by $\mathcal{D}$, then $s_0 \xrightarrow{e'_1} s_1 \xrightarrow{e'_2} s_2 \dots s_q \xrightarrow{e_r} s_q \dots s_{n-1} \xrightarrow{e'_n} s_n \dots$, where e'_i and e_r (inserted by the unfolding algorithm) correspond to e_i and e_r, respectively, is recognized by $\mathcal{A}$ and vice versa.

Lemma 1. *When the configuration of $\mathcal{D}$ is A_i (i.e., $\texttt{cfg} = i$), and $s \xrightarrow{e} s'$ is a possible step in A_i, then $s \xrightarrow{e'} s'$, where e' is an image of e in $\mathcal{A}$, is a possible step in $\mathcal{A}$ and vice versa.*

Lemma 2. *When the configuration of $\mathcal{D}$ is A_i (i.e., $\texttt{cfg} = i$), and rule r applies to A_i at location s, then $s \xrightarrow{e_r} s$, where e_r is an image of r in $\mathcal{A}$, is a possible step in $\mathcal{A}$ and vice versa.*

Proof. We start by proving Lemma 1. There are two cases to consider. First, if edge e is present in all configurations of $\mathcal{D}$, then e and its representation e' are identical. Second, if edge e is not present in all configurations of $\mathcal{D}$ (i.e., it disappears in certain configurations), then the proposed algorithm computes the preconditions of e' according to those of e and where e appears. In other words, if $g = (s, g_c, g_v)$ is the precondition of edge e, then $g' = (s, g_c, \texttt{le\&\&}g_v)$ is the precondition of its representation e', where `le` is valuated true only when the current configuration of $\mathcal{A}$ is the one in which e appears. Hence, if $s \xrightarrow{e} s'$ is a possible step in A_i, then $s \xrightarrow{e'} s'$ is a possible step in $\mathcal{A}$ and vice versa. It is important to note that the postconditions of both e and e' are the same.

Regarding Lemma 2, if an application of rule $r = \langle L \xleftarrow{\varphi_l} I \xrightarrow{\varphi_r} R, (s, g_c, g_v), (\delta, \alpha) \rangle$ from configuration A^s at location s to configuration A^t is represented by edge e_r, then the following holds true by definition. (i) The preconditions of e_r are $g' = (s, g_c, \texttt{"cfg==i\&\&}g_v\texttt{"})$, such that `i` is the index of A^s. (ii) The postconditions of e_r are $u' = (\delta, \texttt{"}\alpha\texttt{,cfg:=j"})$, such that `j` is the index of A^t. Consequently, if rule r applies to A^s at location s under preconditions g_c and g_v, then $s \xrightarrow{e_r} s$ is a possible step in $\mathcal{A}$, with preconditions $(s, g_c, \texttt{"cfg==i\&\&}g_v\texttt{"})$. This relationship also holds true in the reverse direction. □

Theorem 1. *If a run $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \dots s_q \xrightarrow{r} s_q \dots s_{n-1} \xrightarrow{e_n} s_n \dots$ is recognized by $\mathcal{D}$, then $s_0 \xrightarrow{e'_1} s_1 \xrightarrow{e'_2} s_2 \dots s_q \xrightarrow{e_r} s_q \dots s_{n-1} \xrightarrow{e'_n} s_n \dots$ is recognized by the unfolding $\mathcal{A}$ and vice-versa.*

Proof. From Lemmas 1 and 2, we can conclude that if a run $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \dots s_q \xrightarrow{r} s_q \dots s_{n-1} \xrightarrow{e_n} s_n \dots$ is recognized by $\mathcal{D}$, then the equivalent run $s_0 \xrightarrow{e'_1} s_1 \xrightarrow{e'_2} s_2 \dots s_q \xrightarrow{e_r} s_q \dots s_{n-1} \xrightarrow{e'_n} s_n \dots$ is recognized by its unfolding $\mathcal{A}$ and vice-versa. □

7. Illustrative Example

In this section, we present a description of a reconfigurable system, demonstrate the use of the proposed formalism in modeling it, and finally, apply the unfolding algorithm.

We consider a job shop that consists of m machines sharing t tools to manufacture two types of products, A and B, from simple components, with $m > t$. The machines select

some components and determine whether to produce either an object of type A, B, or A' using the available tools.

Initially, the two product types, A and B, are being produced. Each machine is in the `idle` state and each tool is in the `free` state. If a machine decides to produce a product of type A, it picks up one tool and begins production, which should take no more than five time units. For products of type B, a machine requires two tools and takes 20 time units to complete production.

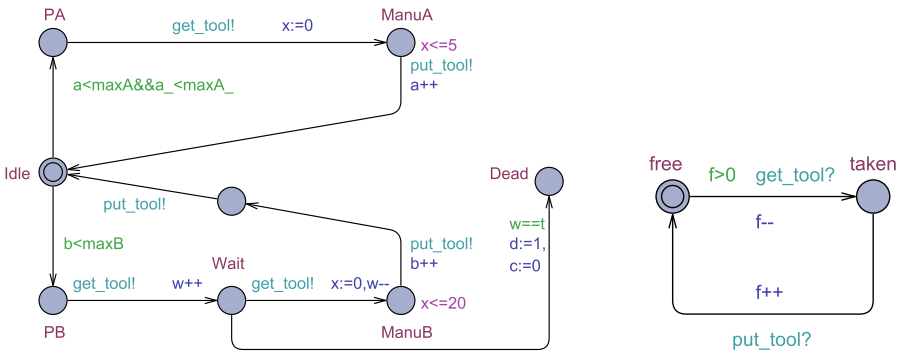
Due to the shared use of tools among the machines, the job shop is prone to deadlocks. For example, if a certain number of machines decide to produce B products, each machine picks up only one tool, no other tools are available, and other machines remain idle, then the job shop is deadlocked. In such situations, the job shop is reconfigured into recovery mode, such that:

- Each idle machine can only start manufacturing a product of type A' without requiring any tools;
- At least one of the machines that decided to produce a B product returns a picked tool and begins production of type A';
- The remaining machines continue to produce type B and once finished, start manufacturing a product of type A'.

Once all tools are available, the job shop returns to its normal mode.

Consider the network of timed automata illustrated in Figure 6, which models the job shop described above. The following points characterize the network:

1. The initial configuration A_0 of machines and a model of tools are depicted in Figure 6a,b, respectively;
2. The synchronization channels `get_tool` and `put_tool` are present;
3. Local clock `x` is used;
4. Constant `t` represents the number of tools;
5. Constants `maxA`, `maxB` and `maxA_` correspond to the maximum number of products A, B, and A' to be manufactured, respectively;
6. Local variables `a`, `b`, and `a_` and global variables `f` and `w` are used to store the number of manufactured products A, B, and A', available tools, and waiting machines for a second tool, respectively;
7. We distinguish a variable `d` used to indicate the presence of a deadlock;
8. The use of local variable `c` is explained later.

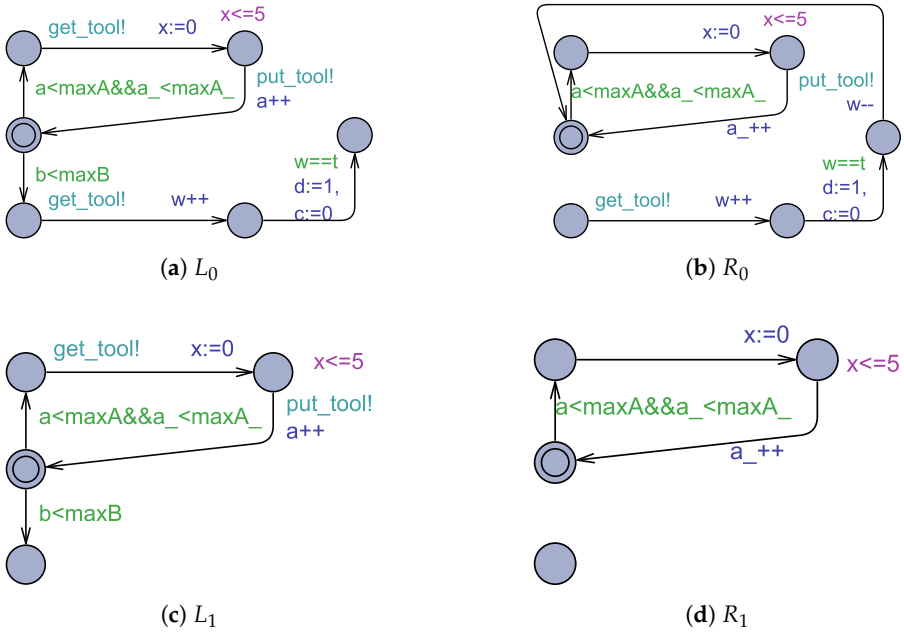


(**a**) A machine (**b**) A tool

Figure 6. A network of timed automata modeling a reconfigurable job shop.

When the value of `w` equals `t`, i.e., the number of waiting machines for a second tool equals the number of tools, the job shop is in a deadlock state. In this scenario, any waiting machine can identify the presence of a deadlock. This is modeled by an edge from location `Waiting` to location `Dead` (see Figure 6a) with the guard "`w==t`" and the effect "`d:=1,c:=0`". Recall that variable `d` is used to indicate the presence of a deadlock state.

Once the value of d becomes one, the job shop must be reconfigured into the recovery mode. First, we apply reconfiguration r_0 illustrated in Figure 7 to the machine which detected the presence of a deadlock, such that its precondition is $g_{r_0} = (\texttt{Dead}, \texttt{true}, \texttt{true})$. The obtained configuration is shown in Figure 8a.



(**a**) L_0 (**b**) R_0

(**c**) L_1 (**d**) R_1

Figure 7. Transformation rules r_0 and r_1.

Remark 1. *Since the left- and right-hand sides and the interface of any rule have the same location set, we omit the interfaces of the rules in the figures presented in this section.*

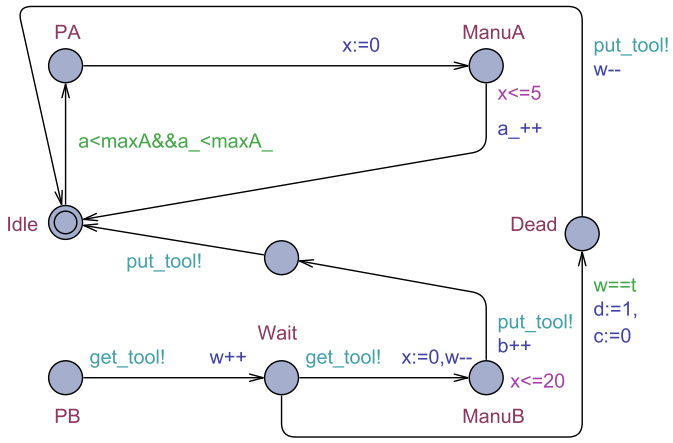
For the rest of the machines, we apply reconfiguration rule r_1 depicted in Figure 7 to each idle machine, whose precondition is $g_{r_1} = (\texttt{Idle}, \texttt{true}, \texttt{d==1})$. The obtained configuration is presented in Figure 8a.

Note that when r_0 is applied to a machine, it can move from location Dead to location Idle (see Figure 8a), and then release the tool. Then, any waiting machine can use that tool to resume the production of product B. After finishing, reconfiguration rule r_1 is applied to it.

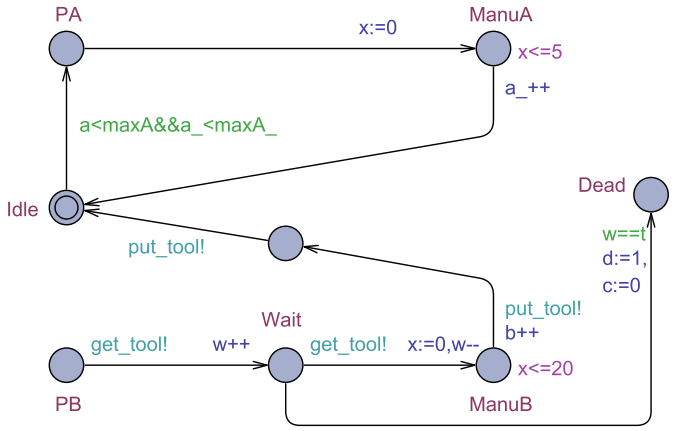
When all tools are available, i.e., f==t, the job shop can return to its initial configuration by applying rules r_2 and r_3 shown in Figure 9. Rule r_2 is applicable to configuration C_1 whose pre- and postconditions are $g_{r_2} = (\texttt{Idle}, \texttt{true}, \texttt{f==t||d==0})$ and $u_{r_3} = (\emptyset, \texttt{"d:=0,c:=0"})$, respectively. However, rule r_3 applies to configuration C_2 whose pre- and postconditions are $g_{r_3} = (\texttt{Idle}, \texttt{true}, \texttt{(f==t||d==0)\&\&c!=0})$ and $u_{r_3} = (\emptyset, \texttt{d:=0})$. Here, variable c is used to prevent the application of r_3 to a machine that has detected the presence of deadlock before applying r_2. This is to ensure that r_2 only applies if r_3 has not been applied first.

Now, we apply the unfolding algorithm to DTA $\mathcal{D} = \langle A_0, \{r_0, r_1, r_2, r_3\}\rangle$, where initial configuration A_0 is shown in Figure 6a. The obtained TA $\mathcal{A}$ illustrated in Figure 10 is semantically equivalent to $\mathcal{D}$.

Finally, to verify some properties of the job shop, we use UPPAAL (i) to create an NTA consisting of three machines m_1, m_2, and m_3 and two tools t_1 and t_2; (ii) and then to check certain properties. The obtained results are demonstrated in Table 1.

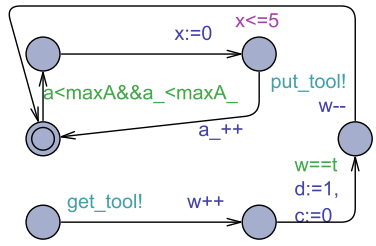


(**a**) Configuration C_1

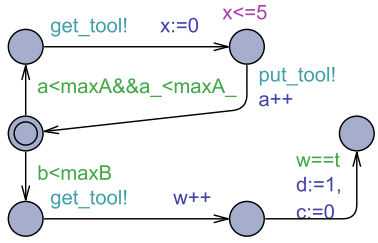


(**b**) Configuration C_2

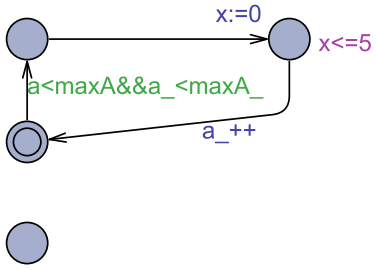
Figure 8. Configurations C_1 and C_2.



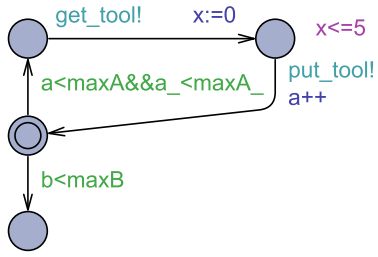
(**a**) L_2



(**b**) R_2



(**c**) L_3



(**d**) R_3

Figure 9. Transformation rules r_2 and r_3.

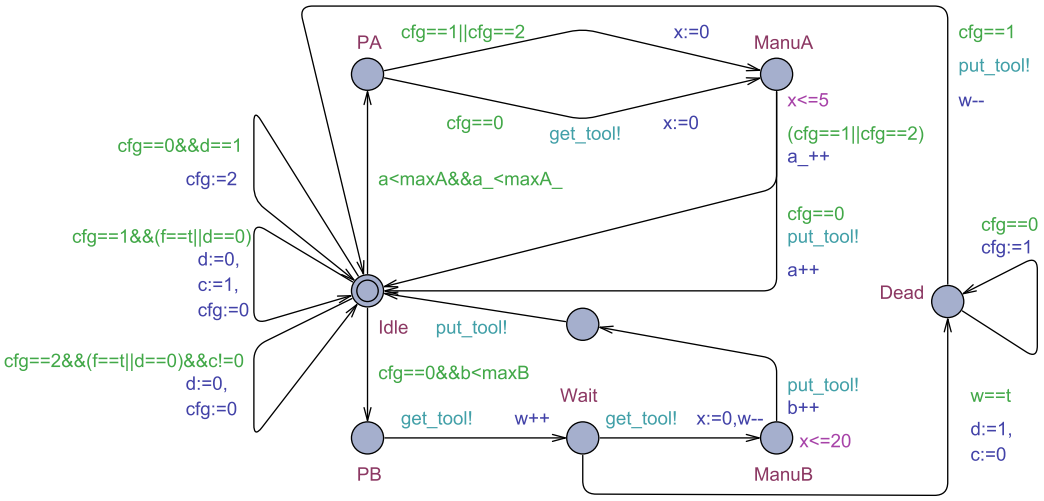


Figure 10. TA $\mathcal{A}$ semantic-equivalent to $\mathcal{D}$.

Table 1. Verification of some properties of interest.

Property	Meaning
`A<> m1.cfg==0 && m2.cfg==0 && m3.cfg==0`	The job shop can always return to its initial mode
`A<> m1.Idle && m2.Idle && m3.Idle`	The job shop can always return to its initial state
`E<> m1.cfg!=0 && m2.cfg!=0 && m3.cfg!=0`	A state in which m_1, m_2, and m_3 are either in C_1 or C_2 is reachable
`m1.Dead -> m1.cfg!=0`	Whenever m_1 reaches location `Dead`, then it can always change its configuration
`A[] not (m1.cfg==2 && m2.cfg==2 && m3.cfg==2)`	A state in which all machines are in C_2 at once is never reachable
`E<> m1.cfg==1 && m2.cfg==1 && m3.cfg==1`	A state in which all machines are in C_1 at once is reachable
`A[] m1.a<=maxA && m1.b<=maxB && m1.a_<=maxA_`	m_1 never exceeds the production limit
`t1.taken -> t1.free`	A tool currently used by a machine will always be free after a while

Remark 2. *To ensure that a TA does not remain in a location indefinitely during model-checking, locations* `PA`*,* `PB`*,* `Wait`*, and* `Dead` *are set as urgent. This means that an automaton must move immediately from these locations whenever possible. In fact, urgent locations can be expressed using only clocks. Hence, no modifications are needed to the proposed formalism level to support this feature.*

8. Conclusions

Modern systems are designed with reconfigurable structures and a high flexibility to meet various complex requirements while maintaining cost-effectiveness. This creates a challenging issue in developing such systems and requires the use of rigorous tools such as timed automata.

The integration of graph transformation systems into formal methods brings several benefits. Nevertheless, several properties that need to be verified by designers become undecidable. To the best of our knowledge, there is no existing work that empowers timed automata by graph transformation systems to model dynamic structures.

In this paper, we presented an approach involving the transformation of timed automata. We leveraged the theory of the double-pushout approach to formulate the transformation rules, resulting in a new formalism called dynamic TA (DTA). Furthermore, we proposed an algorithm that transformed DTA into semantic-equivalent TA. This aimed to enable the use of existing TA analysis tools for the DTA analysis.

In future work, we will exploit the nature of the underlying models to ensure the preservation of global properties of TA models. This will allow a reduction of the temporal

and spatial complexities of the verification process. Additionally, we will consider the relationship between rules, such as a critical pair analysis and compositional rule application, to provide more assistance to designers at the modeling step.

Author Contributions: Conceptualization, S.T., F.G. and N.H.; methodology, S.T., L.K. and N.H.; writing—original draft preparation, S.T. and F.G.; writing—review and editing, S.T., F.G., L.K. and N.H.; supervision, M.K. and M.A.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dima, A.; Bugheanu, A.M.; Boghian, R.; Madsen, D.O. Mapping Knowledge Area Analysis in E-Learning Systems Based on Cloud Computing. *Electronics* **2023**, *12*, 62. [CrossRef]
2. Souri, A.; Rahmani, A.M.; Navimipour, N.J.; Rezaei, R. A hybrid formal verification approach for QoS-aware multi-cloud service composition. *Clust. Comput.* **2020**, *23*, 2453–2470. [CrossRef]
3. Jasim, A.M.; Jasim, B.H.; Neagu, B.C.; Alhasnawi, B.N. Efficient Optimization Algorithm-Based Demand-Side Management Program for Smart Grid Residential Load. *Axioms* **2023**, *12*, 33. [CrossRef]
4. Ecer, F.; Böyükaslan, A.; Hashemkhani Zolfani, S. Evaluation of Cryptocurrencies for Investment Decisions in the Era of Industry 4.0: A Borda Count-Based Intuitionistic Fuzzy Set Extensions EDAS-MAIRCA-MARCOS Multi-Criteria Methodology. *Axioms* **2022**, *11*, 404. [CrossRef]
5. Souri, A.; Ghobaei-Arani, M. Cloud manufacturing service composition in IoT applications: a formal verification-based approach. *Multimed. Tools Appl.* **2022**, *81*, 26759–26778. [CrossRef]
6. Awan, K.A.; Ud Din, I.; Almogren, A.; Khattak, H.A.; Rodrigues, J.J.P.C. EdgeTrust: A Lightweight Data-Centric Trust Management Approach for IoT-Based Healthcare 4.0. *Electronics* **2023**, *12*, 140. [CrossRef]
7. El Ballouli, R.; Bensalem, S.; Bozga, M.; Sifakis, J. Programming dynamic reconfigurable systems. *Int. J. Softw. Tools Technol. Transf.* **2021**, *23*, 701–719. [CrossRef]
8. Alur, R.; Dill, D. Automata for modeling real-time systems. In *Automata, Languages and Programming, Proceedings of the 17th International Colloquium, Warwick University, UK, 16–20 July 1990* ; Paterson, M.S., Ed.; Springer: Berlin/Heidelberg, Germany, 1990; pp. 322–335.
9. Souri, A.; Rahmani, A.M.; Jafari Navimipour, N. Formal verification approaches in the web service composition: A comprehensive analysis of the current challenges for future research. *Int. J. Commun. Syst.* **2018**, *31*, e3808. [CrossRef]
10. Vaandrager, F. A First Introduction to UPPAAL. In *Industrial Handbook*; 2011; pp. 18–48. Available online: https://www.researchgate.net/publication/228919420_A_First_Introduction_to_Uppaal (accessed on 30 December 2022).
11. Behrmann, G.; David, A.; Larsen, K.G. A tutorial on UPPAAL. In Proceedings of the Formal Methods for the Design of Real-Time Systems, Bertinoro, Italy, 13–18 September 2004; Springer: Berlin/Heidelberg, Germany, 2004; pp. 200–236.
12. Chan, C.C.; Yang, C.Z.; Fan, C.F. Security Verification for Cyber-Physical Systems Using Model Checking. *IEEE Access* **2021**, *9*, 75169–75186. [CrossRef]
13. Valero, V.; Díaz, G.; Cambronero, M.E. Timed Automata Modeling and Verification for Publish-Subscribe Structures Using Distributed Resources. *IEEE Trans. Softw. Eng.* **2017**, *43*, 76–99. [CrossRef]
14. Lin, Q.Q.; Wang, S.L.; Zhan, B.H.; Gu, B. Modelling and verification of real-time publish and subscribe protocol using UPPAAL and Simulink/Stateflow. *J. Comput. Sci. Technol.* **2020**, *35*, 1324–1342. [CrossRef]
15. Moussa, B.; Kassouf, M.; Hadjidj, R.; Debbabi, M.; Assi, C. An Extension to the Precision Time Protocol (PTP) to Enable the Detection of Cyber Attacks. *IEEE Trans. Ind. Inform.* **2020**, *16*, 18–27. [CrossRef]
16. Mouelhi, S.; Laarouchi, M.E.; Cancila, D.; Chaouchi, H. Predictive Formal Analysis of Resilience in Cyber-Physical Systems. *IEEE Access* **2019**, *7*, 33741–33758. [CrossRef]
17. Murata, T. Petri nets: Properties, analysis and applications. *Proc. IEEE* **1989**, *77*, 541–580. [CrossRef]
18. Kulcsár, G.; Lochau, M.; Schürr, A. Graph-rewriting Petri nets. In Proceedings of the International Conference on Graph Transformation (ICGT 2018), Toulouse, France, 25–26 June 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 79–96.
19. Tigane, S.; Kahloul, L.; Benharzallah, S.; Baarir, S.; Bourekkache, S. Reconfigurable GSPNs: A modeling formalism of evolvable discrete-event systems. *Sci. Comput. Program.* **2019**, *183*, 102302. [CrossRef]
20. Wang, H.; Wu, J.; Zhu, X.; Chen, Y.; Zhang, C. Time-Variant Graph Classification. *IEEE Trans. Syst. Man Cybern. Syst.* **2020**, *50*, 2883–2896. [CrossRef]
21. Tigane, S.; Kahloul, L.; Hamani, N.; Khalgui, M.; Ali, M.A. On Quantitative Properties Preservation in Reconfigurable Generalized Stochastic Petri Nets. *IEEE Trans. Syst. Man Cybern. Syst.* **2022**, *early access*. [CrossRef]

22. Heckel, R.; Küster, J.M.; Taentzer, G. Confluence of Typed Attributed Graph Transformation Systems. In *Proceedings of the Graph Transformation*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 161–176.
23. Jayaraman, P.; Whittle, J.; Elkhodary, A.M.; Gomaa, H. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In Proceedings of the Model Driven Engineering Languages and Systems, Nashville, TN, USA, 30 September–5 October 2007; Springer: Berlin/Heidelberg, Germany, 2007; pp. 151–165.
24. Taentzer, G. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Proceedings of the Applications of Graph Transformations with Industrial Relevance, Charlottesville, VA, USA, 27 September–1 October 2003; Springer: Berlin/Heidelberg, Germany, 2004; pp. 446–453.
25. Göttmann, H.; Luthmann, L.; Lochau, M.; Schürr, A. Real-Time-Aware Reconfiguration Decisions for Dynamic Software Product Lines. In Proceedings of the 24th ACM Conference on Systems and Software Product Line, Montreal, QC, Canada, 19–23 October 2020; Association for Computing Machinery: New York, NY, USA, 2020. [CrossRef]
26. Göttmann, H.; Bacher, I.; Gottwald, N.; Lochau, M. Static Analysis Techniques for Efficient Consistency Checking of Real-Time-Aware DSPL Specifications. In Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '21), Krems, Austria, 9–11 February 2021; Association for Computing Machinery: New York, NY, USA, 2021. [CrossRef]
27. Zhou, W.; Liu, L.; Lü, S.; Zhang, P. Toward Formal Modeling and Verification of Resource Provisioning as a Service in Cloud. *IEEE Access* **2019**, *7*, 26721–26730. [CrossRef]
28. Aman, B.; Ciobanu, G. Dynamics of reputation in mobile agents systems and weighted timed automata. *Inf. Comput.* **2022**, *282*, 104653. [CrossRef]
29. Alur, R.; Henzinger, T.A.; Vardi, M.Y. Parametric real-time reasoning. In Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, San Diego, CA, USA, 16–18 May 1993; pp. 592–601.
30. Bundala, D.; Ouaknine, J. Advances in Parametric Real-Time Reasoning. In Proceedings of the Mathematical Foundations of Computer Science 2014, Budapest, Hungary, 26–29 August 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 123–134.
31. Cordy, M.; Schobbens, P.Y.; Heymans, P.; Legay, A. Behavioural Modelling and Verification of Real-Time Software Product Lines. In Proceedings of the 16th International Software Product Line Conference, Beijing, China, 16–23 September 2016; Association for Computing Machinery: New York, NY, USA, 2012; Volume 1, pp. 66–75. [CrossRef]
32. Luthmann, L.; Stephan, A.; Bürdek, J.; Lochau, M. Modeling and Testing Product Lines with Unbounded Parametric Real-Time Constraints. In Proceedings of the 21st International Systems and Software Product Line Conference (SPLC '17), Sevilla, Spain, 25–29 September 2017; Association for Computing Machinery: New York, NY, USA, 2017; Volume A, pp. 104–113. [CrossRef]
33. Luthmann, L.; Gerecht, T.; Stephan, A.; Bürdek, J.; Lochau, M. Minimum/maximum delay testing of product lines with unbounded parametric real-time constraints. *J. Syst. Softw.* **2019**, *149*, 535–553. [CrossRef]
34. Bürdek, J.; Lochau, M.; Bauregger, S.; Holzer, A.; von Rhein, A.; Apel, S.; Beyer, D. Facilitating Reuse in Multi-goal Test-Suite Generation for Software Product Lines. In Proceedings of the Fundamental Approaches to Software Engineering, London, UK, 11–18 April 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 84–99.
35. Latreche, F.; Belala, F. RDTA: Recursive and Dynamic Timed Automata for Web Services Composition Analysis. *Int. J. Embed.-Real-Time Commun. Syst. (IJERTCS)* **2014**, *5*, 42–67. [CrossRef]
36. Campana, S.; Spalazzi, L.; Spegni, F. Dynamic Networks of Timed Automata for collaborative systems: A network monitoring case study. In Proceedings of the 2010 International Symposium on Collaborative Technologies and Systems, Chicago, IL, USA, 17–21 May 2010; pp. 113–122. [CrossRef]
37. Attie, P.C.; Lynch, N.A. Dynamic input/output automata: A formal and compositional model for dynamic systems. *Inf. Comput.* **2016**, *249*, 28–75. [CrossRef]
38. Bettira, R.; Kahloul, L.; Khalgui, M.; Li, Z. Reconfigurable Hierarchical Timed Automata: Modeling and Stochastic Verification. In Proceedings of the 2019 IEEE International Conference on Systems, Man and Cybernetics (SMC), Bari, Italy, 6–9 October 2019; pp. 2364–2371.
39. Bettira, R.; Kahloul, L.; Khalgui, M. A Novel Approach for Repairing Reconfigurable Hierarchical Timed Automata. In Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020), online, 5–6 May 2020; pp. 398–406.
40. Tigane, S.; Kahloul, L.; Baarir, S.; Bourekkache, S. Dynamic GSPNs: Formal Definition, Transformation towards GSPNs and Formal Verification. In Proceedings of the 13th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '20), Tsukuba, Japan, 18–20 May 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 164—171.