Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01205-w

REGULAR PAPER


# A model template for reachability-based containment checking of imprecise observations in timed automata

**Sascha Lehmann**[1] · **Sibylle Schupp**[1]

Received: 14 July 2023 / Revised: 11 May 2024 / Accepted: 30 July 2024


**Abstract**
Verifying safety requirements by model checking becomes increasingly important for safety-critical applications. For the validity of such proof in practice, the model needs to capture the actual behavior of the real system, which could be tested by containment checks of real observation traces. Basic equivalence checks, however, are not applicable if the system is only partially or imprecisely observable, if the model abstracts from explicit states with symbolic semantics, or if the checks are not expressible in the logics supported by a model checker. In this article, we solve the problem of observation containment checking in timed automata via reachability checking on tester systems. We introduce the logic *SRL (sequence reachability logic)* to express observations as sequences of delayed reachability properties. Through *SBLL* (introduced by Aceto et al.) as intermediate logic, we synthesize a set of matcher model templates for partial and imprecise observations and further extend these templates for the case of limited state accessibility in a model. For the obtained matching traces, we define the back-transformation into the original model domain and formally prove the correctness of the transformation. We implemented the observation matching approach, and apply it to a set of 7 demo and 3 case study models with different levels of observability. The results show that all positive and negative observations are correctly classified, and that the most advanced matcher model instance still offers average run times between 0.1 and 1 s in all but 3 scenarios.

**Keywords** Containment checking · Reachability problem · Model transformation · Timed automata · Observation matching

## 1 Introduction

The trustworthiness of model checking generally depends on how well a model captures reality. For cyber-physical systems (CPS), reality exhibits phenomena such as noise or missing data. System models as well as observations abstract from that reality and, thus, cannot capture the behavior and complexity of a real system in an exact manner—a problem commonly known as the modeling gap. A system model of a moving car, for example, might assume an idealized constant velocity after acceleration, while the real velocity may show natural fluctuation, and the measurement of said velocity adds further deviations due to limited sensor and data precision. However, if the runs of a model and a particular sequence of real-world observations do not match, the model is not necessarily of poor quality and unsuitable for property checking. By shifting or coarsening the observation sequence, and allowing for the absence of particular variable data in individual observations, it might often still be possible to find a match with runs of the model, so that, under consideration of such imprecisions, one could continue to trust the model. Then, the model only has to be dismissed if it, despite the allowances, does not match reality; the particular tolerances can be customized by formulating the model checking query accordingly.

Given a sequence of real-world observations that does not exactly match with any of the runs of its model, we are concerned in this article with designing, verifying, and implementing an automated, constructive approach that decides whether, in the presence of imprecisions, the real-world observation sequence can be transformed to a run of the model. We call this problem the "containment checking" problem and answer it for timed automata (TAs), which

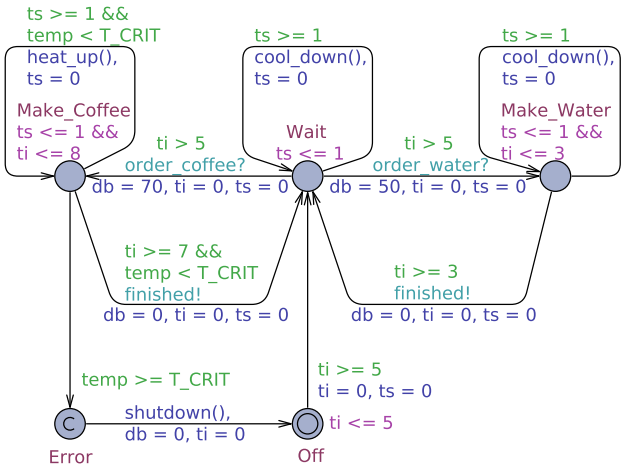
Communicated by Andrzej Wasowski.

✉ Sascha Lehmann
s.lehmann@tuhh.de

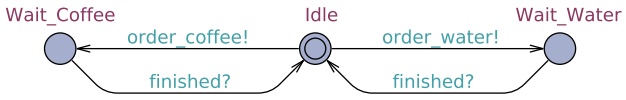
Sibylle Schupp
schupp@tuhh.de

[1] Institute for Software Systems, Hamburg University of Technology, Am Schwarzenberg-Campus 3 (E), 21073 Hamburg, Germany

Published online: 19 September 2024




(a) The vending machine model




(b) The user model

**Fig. 1** The example model of a vending machine

are a common choice for model checking of cyber-physical systems. The literature, especially in the related domain of real-time logics and pattern specifications, puts a strong focus on containment checks of timed sequences of actions, but—to the best of our knowledge—investigates neither state sequences nor the imprecisions that may cause a gap between the modeled behavior and such sequences. For containment checks of (imprecise) observations in TAs, a challenge is to make runs in the continuous-time semantics of the system comparable to observations that are based on pointwise (i.e., discrete) semantics. The following introductory example illustrates the containment checking problem and the concepts of matches and mismatches:

***Example 1*** (The running example model) The system shown in Fig. 1 consists of a vending machine (Fig. 1a) and a user (Fig. 1b) who orders beverages. The vending machine models the normal locations `Off`, `Wait`, `Make_Coffee`, and `Make_Water`, the corresponding levels of sound (`db`) and temperature (`temp`), and an `Error` state, which is reached whenever the temperature exceeds a safe bound (`T_CRIT`) during coffee preparation, resulting in a shutdown of the machine. A user can either `order_coffee` or `order_water` and gets informed once the beverage preparation is `finished`.

If observations were exact, we could for example observe the following sequence of timed states via sensors:

$$\begin{aligned} obs = \langle &(0s, \mathit{Off}, 0db, 20\ ^\circ\mathrm{C}),\\ &(12s, \mathit{Make\_Water}, 50db, 20^\circ\mathrm{C}),\\ &(25s, \mathit{Make\_Coffee}, 70db, 40^\circ\mathrm{C})\rangle, \end{aligned} \tag{1}$$

or get the following sequence of timed actions by observing the interaction of a user with the machine:

$$obs = \langle (11s, \texttt{order\_water}), (14s, \texttt{finished})\rangle. \tag{2}$$

Both observation sequences directly match, e.g., the following model run for a global clock $t$:

$$\begin{aligned} r = &\langle \texttt{Off}, \{(\texttt{t}, 0)\}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle\\ &\rightarrow \langle \texttt{Wait}, \{(\texttt{t}, 6)\}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle\\ &\rightarrow \langle \texttt{Make\_}\mathit{Water}, \{(\texttt{t}, 12)\}, \{(\texttt{db}, 50), (\texttt{temp}, 20)\}\rangle\\ &\rightarrow \langle \texttt{Wait}, \{(\texttt{t}, 15)\}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle\\ &\rightarrow \langle \texttt{Make\_}\mathit{Coffee}, \{(\texttt{t}, 25)\}, \{(\texttt{db}, 70), (\texttt{temp}, 40)\}\rangle. \end{aligned} \tag{3}$$

In reality, however, observations are not ideal, so that, e.g., the temperature and sound levels measured via sensors may exhibit noise due to measurement uncertainty, temperature data may not be measured while water is prepared, and the measurement may have only started after 3 s into startup. A sequence of observed states might thus look as follows:

$$\begin{aligned} obs = \langle &(0s, \mathit{Off}, 0\ db, 19^\circ\mathrm{C}),\\ &(9s, \mathit{Make\_Water}, 48\ db, \_),\\ &(22s, \mathit{Make\_Coffee}, 72\ db, 41^\circ\mathrm{C})\rangle, \end{aligned} \tag{4}$$

Intuitively, one would argue that the observation sequence in Eq. (4) still conceptually matches the run in Eq. (3), but model checking based on exact runs would classify the observation as a mismatch. Therefore, we require an approach that allows partial observations and slight deviations in measured variable data, and would identify Eq. (3) as suitable match. We develop such an approach in the scope of this article. That way, we could conclude that the model is still suitable, e.g., for deriving statements on the potential futures of the runs we identified as matching, such as "in the following $x$ time units, the current delay enforced during `Wait` is still sufficient to cool down the machine enough to render the `Error` state unreachable".

Our main idea is to express an observation sequence together with its allowed imprecisions as logical formula, and translate the model and formula into an equivalent reachability problem on a corresponding matcher system. In real-time model checking, reachability testing is commonly used to verify arbitrary formulae expressed in a real-time logic by transforming the formula into a *test automaton* (in the real-time context usually called *monitor* or *observer*), composing the test automaton with the original model, and checking if a dedicated rejection state of the test automaton is unreachable

in the composition [1]. Using the approach we introduce in this article, we can perform the checking of imprecise observations directly on the model, as both the observations and the allowed imprecisions are integrated into the derived test automaton, where an acceptance state is only reached by a run if all constraints imposed by the observations are fulfilled. That way, we can also make use of the optimizations for reachability checking commonly implemented in model checkers.

For the observation formulae, we introduce the helper logic *SRL (sequence reachability logic)*, which allows expressing sequences of reachability properties with conditions on clock times, variables, and actions (viz. to express (im-)precise observations), and intermediate time delays (viz. the time durations between the observed data points) relative to fixed reference clocks (viz. for potential shifts between model and observation time). We then show that $SRL$ formulae can be transformed into formulae of *SBLL (safety and bounded liveness logic)*, a logic introduced by Aceto et al. [2], as the formulae of the logics constitute duals via negation. The transformation of $SBLL$ formulae into test automata is already well-defined. Individually, the two logics serve different purposes: The work of Aceto et al. deals with global properties over all paths and focuses on the expressivity boundaries of $SBLL$ as testable logic, while $SRL$ deals with matching properties on individual paths and focuses on the expression of imprecise observations. But the dual relation between both logics allows us to combine the more natural specification of observations in $SRL$ with the straight-forward reduction to reachability problems in $SBLL$.

In summary, this article makes the following 4 contributions:

1. We solve the problem of observation containment checking for timed automata via reachability checking.
2. We introduce the helper logic $SRL$ (as derivation and dual of $SBLL$) for intuitive formulation of sequences of reachability properties, and formulate $SRL$ properties for the cases of deviating, partial, shifted, and qualitative observations.
3. We abstract matching to 4 generally applicable matcher models for the cases expressed by the $SRL$ properties (2 templates) and the separate case of matching intermediate states (2 templates) and provide a solution to transform the resulting runs of the matcher model to the original model domain.
4. We provide an implementation of the matching approach for containment checking of observation data in *Uppaal* [3] TA models, and perform a series of experiments on a suite of given case study and demo models, for which we analyze the viability and performance of matching both originally observed (via simulation of the original model) and altered runs (via semi-random data manipulation) of varying precision.

The remainder of this article is structured as follows: We describe the related work in Sect. 2, followed by prerequisite knowledge of timed automata, observations, and a reference approach for reachability checking [2] in Sect. 3. Afterward, we introduce the notion of matching, the synthesis, simplification, and extension steps of the matcher model, and the back-transformation of matching runs in Sect. 4. Then, we cover the tool implementation in Sect. 5, followed by the experiments conducted with the tool in Sect. 6. Finally, we conclude the article in Sect. 7.

## 2 Related work

In this section, we discuss existing work on *test automata* and *observation matching* and give a conceptual overview of behavior preservation and imprecisions related to the steps involved in our observation matching approach as well as existing model checker support.

### 2.1 Test automata

Applications of test automata that transform real-time properties into reachability problems for behavior observation in TAs date back to the 1990s, where the synthesis of observers was usually tailored to practical applications [4]. A more generalized test automata synthesis approach, which also allows composition of properties, is described by Aceto et al. [2], whose proposed logics $SBLL$ (to which we translate in our approach) and $L_{\forall S}$ constitute the limits of expressivity of properties reducible to reachability properties. Sets of patterns for different property classes were introduced for model checking in general [5, 6] (usually for the discrete case) and concretized for various formalisms including real-time temporal logics for timed automata [7], time Petri nets [8], and timed UML models [9]. The identification and formalization of test automata as observers are still actively researched, as the set of real-time requirements in practice keeps growing, and at the present day, [10] provides the most comprehensive catalog for nested real-time property specification (e.g., $AG(P \rightarrow AF^{[0,t]}(S))$, which describes that globally on all paths, if $P$ occurs, $S$ always eventually holds after at most $t$ time units).

In this article, we also synthesize test automata, where the type of "specification patterns" in our case are the real-time properties of observation matching under imprecisions describable in our helper logic $SRL$, which we then reduce to reachability problems. While the literature here focuses on sequences of actions [7], and—originating in real-time systems—deals with safety and liveness properties in the

exact trace space of the model, we extend the scope to sequences of (imprecise) states, dealing with sequences of local reachability problems that model different types of non-exact observations.

### 2.2 Observation matching

Observation matching has been performed in form of pattern matching in several scenarios in the past, including Boolean signal matching [11, 12], and—in more recent studies—as parametric timed pattern matching in parametric timed automata [13]. In theory, observation matching relates to the topics of membership queries [14] and inverse (identification) problems [15], dealing with the questions of whether observations are contained in particular model traces, and which inputs (e.g., for TAs, which action transitions) result in particular observations, respectively. Closely related to our work, [16] performs matching of trace observations on TAs. In that work, the transformation of specification automata for testing is achieved by introducing a dedicated location `Err`, and adding transitions from each original location to that error location for each non-triggerable action and its corresponding time range. Reachability analysis on `Err` then checks if the system behaves correctly.

Our work also reduces properties to reachability analysis. However, instead of explicitly checking for all non-intended behaviors in each location, our test automaton simply performs the check for desired behavior (i.e., whether given observation states are reached in time), leading to intended deadlocks on violating paths, which overall results in a more concise composed matcher system. Furthermore, we provide a dedicated language for the specification of observation matching properties. Finally, we do not focus on timed action traces alone, but also cover sequences of observed states and their potential imprecision.

### 2.3 Behavior preservation and uncertainties

During the model simplification step (cf. Section 4.5), the matcher behavior is required to be preserved. Various works describe such behavior preservation in general [17–19] and in connection with a simultaneous adaption of property formulae [20]. In our case, we ensure that the behavior is preserved in the context of the reachability formula `E<> Matcher.S` by only applying changes that keep the sets of reachable states in each sequential section of the test automaton unchanged.

During the model extension step (cf. Section 4.6), we enable the observation matching process to also consider deviating, partial, and qualitative observations. While the question on similarity between models and traces is usually answered in a binary sense, [21] describes quantified similarity where "closely corresponding" timing of events, i.e., time deviation, is allowed in simulations. Variable data deviations occur naturally in pattern recognition and classification, where (potentially deviating) inputs are assigned to classes [22], leading to the concept of "soft matching" [23]; for observation matching, the classes are the potential (symbolic) runs, which might overlap. Partial observability is approached, among others, by Krichen et al. [24], who perform black-box conformance testing using a timed extension of the IOCO relation, and investigate the opposing concepts of dense time (in continuous systems) and periodic clocks (in discrete systems) for test implementation. In general, the interpretation of observable behavior [25] is crucial for the consideration of partial observations. In terms of actions, partial observability is described, e.g., in [26] for the design of FDIR (fault detection, isolation, and recovery) components, where the set of actions of timed automata is split into observable, unobservable, controllable and uncontrollable actions. Qualitative observations are usually found when a system provides control states, e.g., abstract locations in hybrid systems [27] specifying the currently applied rates.

All aforementioned works assume that uncertainties occur whenever real systems are involved, and these uncertainties have to be taken into account when applying formal methods to observations of such systems. For the case of timed automata, our work thus provides matcher models for common types of observability and observation precision.

### 2.4 Model checker support

In the context of Uppaal [3], two extensions were developed for timed i/o action sequences: Uppaal Tron [28] checks for relativized timed i/o conformance relations, and comes closest to the observation matching approach we provide in this work. However, Uppaal Tron does not support (sequences of) observed variable data, and requires the dedicated programming of custom adapters for the system under test to obtain observations. Furthermore, particular features such as data arrays, broadcast channels, and committed locations are currently experimental only. Our work offers the advantage of providing a concise set of matcher models intended for parallel composition with any original TA model, which allows the integration of matching into existing model checkers. Uppaal ECDAR [29] checks for simulation refinements between processes, which could also be used to check if particular model runs are refinements of a given observation. However, similar to Uppaal Tron, the tool is currently limited to timed sequences of actions, so that our central use case of matching data variables with potential imprecisions would not be supported. For the case of $LTL$ properties in the domain of runtime verification, monitoring of (partially observable) systems was implemented on top of existing model checkers such as `nuXmv` [30].

## 3 Prerequisites

In this section, we provide the model formalism of (extended) timed automata (Sect. 3.1), definitions of observation traces and sequences (Sect. 3.2), and an overview of the base workflow for property and model transformation into reachability problems as introduced by Aceto et al. [2] (Sect. 3.3).

### 3.1 Timed automata

A *timed automaton (TA)* is a state machine extended by (real-valued) clocks and further labeled with constraints and resets of these clocks as well as actions. Clock constraints appear as *edge guards*, which control when an edge can be triggered (e.g., `ti >= 5` on the edge `Off` → `Wait` in Fig. 1a), and *location invariants*, which control for how long a location may remain active (e.g., `ti <= 5` in the location `Off`), and *resets* on edges set particular clocks to 0 (e.g., `ti=0` on `Off` → `Wait`). Formally, a TA is a tuple $\langle \Sigma, L, l_0, C, E, I \rangle$, where $\Sigma$ is a finite set of actions (including *internal*—also called *silent*—actions, which are unobservable from the outside and are either explicitly denoted as $\tau$ or implicitly by leaving out the action label on an edge), $L$ is a finite set of locations, $l_0$ is the initial location, $C$ is a finite set of clocks, $E \subseteq L \times \Sigma \times \Phi_G(C) \times 2^C \times L$ is a set of edges between locations, where each edge has an action $a \in \Sigma$, a guard $g \in \Phi(C)$, and a reset $r \in 2^C$ of a subset of clocks to 0, and $I : L \to \Phi_I(C)$ is a mapping from locations to invariants. $\Phi_G(C)$ contains all possible conjunctions over constraints $t_a <>_G c$ and $t_a - t_b <>_G c$, with $t_a, t_b \in C$, $c \in \mathbb{N}$, and $<>_G \in \{<, \leq, =, \geq, >\}$, and similarly, $\Phi_I(C)$ contains all such constraints $t_a <>_I c$ and $t_a - t_b <>_I c$ with $<>_I \in \{<, \leq\}$.

Our work uses an extended definition of TAs, called *ETA*, which is implemented in Uppaal and adds variable states, different types of locations (*normal*, *urgent*, and *committed*), and synchronization between automata processes composed to *networks of timed automata (NTAs)* (cf. [31]). Formally, an ETA is a tuple $\langle \Sigma, L, l_0, C, E, I, V, L_U, L_C, Ch_{Bi}, Ch_{Br} \rangle$, where $\Sigma$, $L$, $l_0$, $C$, and $I$ are as in the TA definition, $E \subseteq L \times \Sigma \times \Phi(C) \times 2^C \times A(V) \times L$ is a set of edges similar to TAs with additional variable assignments $va \in A(V)$, $V$ is a set of variables, $L_U \subseteq L$ and $L_C \subseteq L$ are sets of urgent and committed locations (with $L_U \cap L_C = \emptyset$), and $Ch_{Bi}$ and $Ch_{Br}$ are binary and broadcast synchronization channels (with $Ch_{Bi} \cup Ch_{Br} \cup \{\tau\} \subseteq \Sigma$ and $Ch_{Bi} \cap Ch_{Br} = \emptyset$). We define $A(V) = [V \rightharpoonup \mathbb{B} \cup \mathbb{Z}]$ as the set of all partial assignment functions $\alpha : V \rightharpoonup \mathbb{B} \cup \mathbb{Z}$, where each $\alpha$ represents a mapping from a subset of variables to Boolean or integer values.

We define the operational semantics of an ETA $M$ similar to [2] (i.e., while adding data variables to their definition) by the transition system $TS(M) = (\Sigma, S, s_0, \to)$, where $\Sigma$ is the set of actions of $M$, $S$ is the set of states $\langle l, u, v \rangle$ of $M$ (where $l$ is a location in $M$, $u$ is a clock valuation, and $v$ is a variable valuation), $s_0$ is the initial state of $M$, and $\to$ is the transition relation, with:

- $(\langle l, u, v \rangle \xrightarrow{\varepsilon(d)} \langle l, u', v \rangle)$ iff $u' = u + d \wedge I(l)(u'), d \in \mathbb{R}_{\geq 0}$
- $(\langle l, u, v \rangle \xrightarrow{a} \langle l', u', v' \rangle)$ iff $\exists e = \langle l, l' \rangle \in E :$ $(ch(e) = a) \wedge g(e)(u) \wedge u' = [r(e)]u \wedge$ $v' = [va(e)]v \wedge I(l')(u')$.

$\xrightarrow{\varepsilon(d)}$ are *delay transitions*, which model time delays $d$ in a particular location and variable state constrained by the location invariant $I(l)(u')$. $\xrightarrow{a}$ are *action transitions*, which model changes between location and variable states enforced by the source location invariant $I(l)(u')$, which triggers the action $a$ constrained by the edge guard $g(e)(u)$ and target location invariant $I(l')(u')$, where $u' = [r(e)]u$ is the resulting clock valuation after applying the resets $r(e)$ to $u$, and likewise, $v' = [va(e)]v$ is the resulting variable state after applying the variable assignments $va(e)$ to $v$.

We define the (possibly infinite) set of runs (i.e., the sequences of states and transitions produced by the transition relation) in $TS(M)$ as $R(M)$, the trace set (i.e., the runs reduced to actions) as $TR(M)$, and the path set (i.e., the runs reduced to states) as $P(M)$. Furthermore, we define for any property $\varphi$ and the initial state $(l_0, u_0, v_0)$ of $TS(M)$:

$$(\mathrm{TS}(M) \models \varphi) \Leftrightarrow ((l_0, u_0, v_0) \models \varphi) \tag{5}$$

Finally, we define $\xrightarrow{\tau}{}^*$ as the reflexive and transitive closure of $\xrightarrow{\tau}$, and define:

- $s \xRightarrow{a} s'$ iff $\exists s'' : s \xrightarrow{\tau}{}^* s'' \xrightarrow{a} s'$
- $s \xRightarrow{\varepsilon(d)} s'$ iff $\exists s = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n = s'$, where $d = \sum\{d_i | \alpha_i = \varepsilon(d_i)\}$.

For a definition of the semantics of networks of timed automata and channel synchronization, see [3].

#### 3.1.1 ETA-specific elements

We explain the elements added by an ETA in the following: *Variable assignments* In an ETA, *variables* can be used to store system data in each state (e.g., variables `temp` and `db` in Fig. 1a). These variables can be re-assigned on edges (e.g., `db = 70` on `Wait` → `Make_Coffee`), and they can be used in edge guards alongside clock constraints to enable edges conditionally (e.g., `temp >= T_CRIT` on `Make_Coffee` → `Error`). In standard Uppaal, variables are restricted to Boolean and integer types.

*Urgent and committed locations* Urgent and committed locations allow splitting atomic sets of actions and state changes over multiple transitions while preserving their atomicity. An *urgent* location (represented in Uppaal by a bottom semicircle resembling a "U") enforces the implicit constraint that no time may pass while the location is active; therefore, processes must not perform delay transitions, whereas action transitions in other processes are still allowed. A *committed* location (represented in Uppaal by a left semicircle resembling a "C") enforces urgency and additionally requires that the location is left in the next possible transition; therefore, no action transitions are allowed in other processes, unless their active locations are committed as well. In Fig. 1a, e.g., atomicity of the temperature check `temp >= T_CRIT` and the instantly initiated `shutdown()` procedure is ensured by the committed `Error` location.

*Binary and broadcast channels* A *binary* channel enables 1-to-1 synchronizations between pairs of edges in two processes, where one process acts as the caller (e.g., `order_coffee!` in Fig. 1b), and the other acts as the listener (e.g., `order_coffee?` in Fig. 1a). Binary channels are blocking for both sides, i.e., a calling edge can only be taken if an edge of another process listens to the same channel, and vice versa; then, both edges are taken simultaneously in the same action transition. A *broadcast* channel enables 1-to-$n$ synchronization between a set of processes, where one process acts as the caller, and $n$ other processes act as listeners (again specified by appending ! and ? to the channel name, respectively). Broadcast channels are only blocking for the listener side, i.e., the number of listeners $n$ may be 0; the edges of the caller and all potential listeners are taken simultaneously in the same action transition.

### 3.1.2 Modeling with timed automata

In practice, particular modeling patterns are applied to implement desired timing behavior, to realize advanced features that are not explicitly covered by the ETA formalism, or to enable certain model transformations. In this work, the following patterns are relevant for simplifying transformations of the matcher model (cf. Section 4.5) and correctly timed synchronization between the original model and the matcher model (cf. Section 4.6.3):

*Timing of triggered edges* The desired timing behavior, i.e., the time range in which an edge can be triggered, is determined by the clock invariants of its source location and its clock guards, whose combination leads to the following 4 cases:

- Omitting both invariants and guards allows the edge to be taken at any time or not at all.
- Using only an invariant (i.e., $t \leq c$) enforces a transition, and does so at the latest at $t = c$.

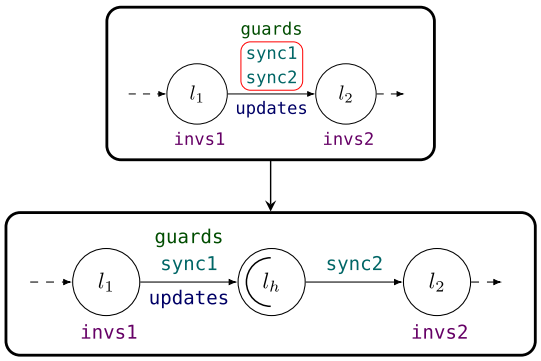



**Fig. 2** Top: multiple synchronizations. Bottom: realization in Uppaal

- Using only guards ensures that the edge is taken at the earliest at $t = d$ (i.e., as lower bound $t \geq d$) or at the latest at $t = d$ (i.e., as upper bound $t \leq d$), but does not enforce a transition.
- Using both invariant and guard (i.e., $t \leq c$ and $t \geq d$) enforces a transition at some time within $[c, d]$; taking the edge exactly at some time $t_e$ is forced by setting $c = d = t_e$, e.g., as done for the edge `Off` $\rightarrow$ `Wait` in Fig. 1a, triggered exactly at `ti == 5`.

*Multiple synchronizations* One advanced feature not explicitly covered by the ETA formalism as implemented by Uppaal is the support for multiple synchronization channels per edge. When a process needs to be synchronized with more than one other process, and each individual synchronization serves a different purpose, one cannot simply use a single broadcast synchronization. A common solution for multiple synchronizations is shown in Fig. 2. The intended effect illustrated in the upper diagram is achieved by splitting the synchronization into two separate edges over an artificial intermediate location $l_h$, which is committed so that the two edges are taken subsequently without interruption. The guards necessarily need to be assigned to the first edge to ensure that the clock constraints are satisfied once the sequence of synchronizations is entered.

*Urgency via explicit clocks* A model transformation we use during simplification of the test automaton is the explicit expression of urgency by a clock $t_h$ that is reset on the incoming edge and constrained to 0 by a location invariant $t_h \leq 0$. Note that such transformation can be applied in both directions (see Fig. 3), i.e., if the clock resets and constraints are given accordingly, a location can be marked urgent instead, removing the corresponding clock constraints.

## 3.2 Observations

The data we match against in this work are *observation sequences* of *timed observation states*, and *observation traces* of *timed actions*. Assume in the following a given

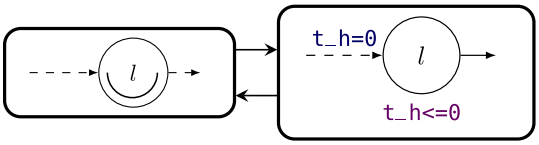



**Fig. 3** Left: modeling urgency. Right: realization in Uppaal via explicit clocks

set of observable system variables $V_{obs} = \{var_1, \ldots, var_k\}$. For state observations, a single timed observation state is a tuple

$$obs := (t, val_1, \ldots, val_k), \tag{6}$$

where $t \in \mathbb{N}$ is the time of the single observation, and $val_1, \ldots, val_k \in (\mathbb{N} \cup NOB)$ are the observed integer values of the corresponding variables $var_1, \ldots, var_k$ if the variable was observed in that time instant, or $NOB$ otherwise. Note that we only consider one single time value per observation state, which corresponds to the global time of the observed real system; local clocks, which are often used as helper constructs for relative timing in TAs, usually have no directly observable equivalent in reality. An observation sequence then has the following form:

$$\begin{aligned} OBS_{seq} := \langle &(t_1, val_{1,1}, \ldots, val_{k,1}), \\ &\ldots, \\ &(t_n, val_{1,n}, \ldots, val_{k,n}) \rangle. \end{aligned} \tag{7}$$

For transition observations, a timed action is defined as pair

$$obs := (t, a), \tag{8}$$

where $t \in \mathbb{N}$ is the observation time as above and $a \in \Sigma$ is a non-silent action. Observation traces (i.e., timed action traces) are then defined as

$$OBS_{tr} := \langle (t_1, a_1), \ldots, (t_n, a_n) \rangle. \tag{9}$$

***Example 2*** An observation sequence of the system in Fig. 1, where the variable `db` was not observed at $0\,s$ and $20\,s$ when the machine was either off or idle, is $OBS_{seq} = \langle (0\,s, NOB), (13\,s, 70\ db), (20\,s, NOB) \rangle$. An observation trace of the same system is $OBS_{tr} = (12\,s, order\_coffee), (19\,s, finished)$.

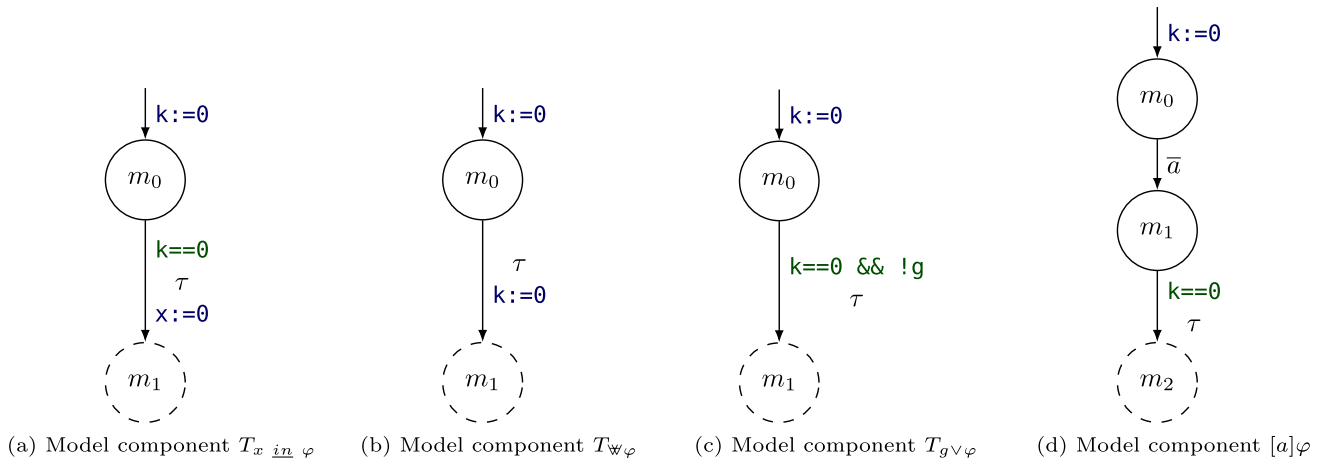
### 3.3 Reachability transformation

Since the introduction of trace-based model checking, model checkers make use of the fact that the individual syntactical constructs of a logic can be represented as automata; for example, model checking $LTL$ properties can be approached by transforming both the model and the negated property into Büchi automata and checking if their intersection is empty. That way, e.g., the property that an atomic proposition $q$ always holds can be represented by an automaton that switches to an "accepting" state as soon as $\neg q$ occurs in a word. In the domain of real-time systems, such *test automata* were introduced as well, which turn more advanced properties (e.g., path-based properties) into reachability checks for efficiency or to enable their checking at all. Aceto et al. [2] introduced a general approach for the reduction of checking a safety or bounded liveness property $\varphi$ (formulated in their $SBLL$ and $L_{\forall S}$ logics) on a timed automaton $M$ to a reachability check on the parallel composition of the transition system $TS(M)$ and the transition system $TS(T_\varphi)$ of a test automaton $T_\varphi$ constructed from $\varphi$. The test automaton $T_\varphi$ is constructed in a way that it contains a reject node $m_T$, which is only reachable in the composition $TS(M) \| TS(T_\varphi)$ if $\varphi$ is not satisfied on $TS(M)$, and conversely, $\varphi$ is satisfied iff $m_T$ is not reachable. In summary, the workflow of their approach consists of the following 5 steps:

1. Formulate a safety or bounded lifeness property $\varphi$ for a model $M$.
2. Compose a test automaton $T_\varphi$ from $\varphi$ based on predefined building blocks.
3. Construct the parallel composition $TS_{pc} = TS(M) \| TS(T_\varphi)$.
4. Check if $m_T$ is unreachable on $TS_{pc}$, i.e., if $\neg reach(TS_{pc}, m_T)$.
5. If so, conclude that $TS(M) \models \varphi$, or otherwise, $TS(M) \not\models \varphi$.

We will show in Sect. 4.2 how the helper logic $SRL$, which we introduce to express observation matching properties, relates to the logic $SBLL$, and how their workflow thus extends to our workflow described in Sect. 4.3, allowing us to solve sequential reachability properties as simple reachability checks.

The authors of [2] introduce a set of building blocks for the construction of the test automaton from the operators of $SBLL$. The subset of building blocks relevant for our approach is shown in Fig. 4, where the clock $k$ is used for local timing in each component, $\tau$ represents silent transitions where no action is observed, and the dashed nodes represent either the starting node of another nested component, or the final reject node $m_T$. Figure 4a shows the component for clock freezing, which sets a dedicated clock $x$ to 0 and thus allows modeling properties with time constraints relative to the time of the freezing operation. Figure 4b shows the component for universal quantification over delays, which allows arbitrary delay in $m_0$ and thus models properties over delayed paths. Figure 4c shows the component for checking clock constraints $g$, which only allows transitioning toward




(a) Model component $T_{x \ in \ \varphi}$ (b) Model component $T_{\forall\varphi}$ (c) Model component $T_{g \vee \varphi}$ (d) Model component $[a]\varphi$

**Fig. 4** The TA building blocks for reachability transformation from [2] used by our approach

the reject location if $g$ is not satisfied, i.e., if $!g$ holds. Likewise, Fig. 4d shows the component for checking properties after an action $a$, which allows transitioning toward the reject location if $a$ is enabled, as only then the satisfaction of the following properties matters; in terms of notation, we write $\overline{a}$ in test automata to refer to the action that synchronizes with $a$ in the parallel composition, i.e., $\overline{a} = a!$ if $a = a?$, and $\overline{a} = a?$ if $a = a!$.

We show in Sects. 4.4 and 4.6 how test automata for observation matching are composed from these components, and thus, how observation matching properties can be reduced to reachability checks on the parallel composition of such test automata and the model of the observed system.

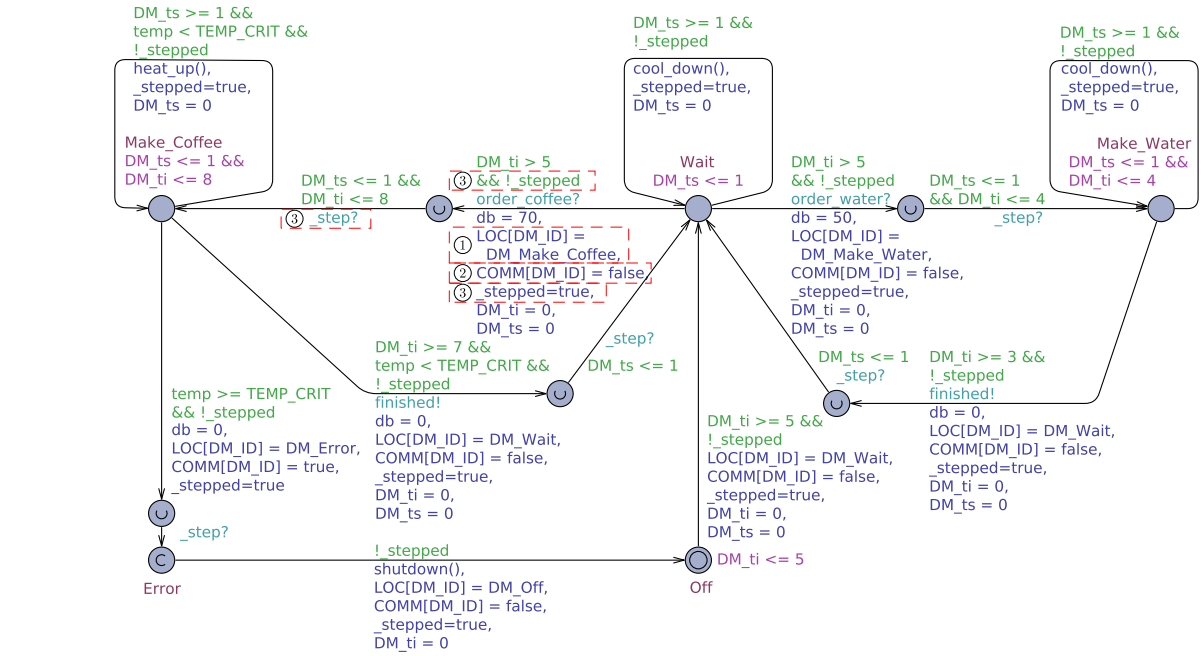
## 4 Matcher model

In this section, we explain the complete cycle of formulating an observation matching property, synthesizing, simplifying and extending the corresponding matcher system, and back-transforming the matching run. First, we explain the underlying concept of matching in Sect. 4.1. To express observation matching as formal property, we then introduce the logic $SRL$ in Sect. 4.2 and show its relation to the logic $SBLL$ (cf. Section 3.3). Based on that relation, we introduce an extension of the original reachability transformation workflow (cf. Section 3.3) in Sect. 4.3, with which we can transform observation matching into a simple reachability problem. Then, we synthesize the base test automaton from the most basic observation matching property in Sect. 4.4, simplify that model in terms of required locations and clocks in Sect. 4.5, and present variants and extensions of the resulting matcher system for the use cases of deviating, partial, delayed, and committed observations in Sect. 4.6. Finally, we explain how a matching result run is transformed back to the domain of the original model in Sect. 4.7. As main result, we derive 4 versions of the matcher template in total, whose integration into the original model—combined with a set of accompanying changes to the original model templates—allows checking for the inclusion of observation data for the aforementioned scenarios.

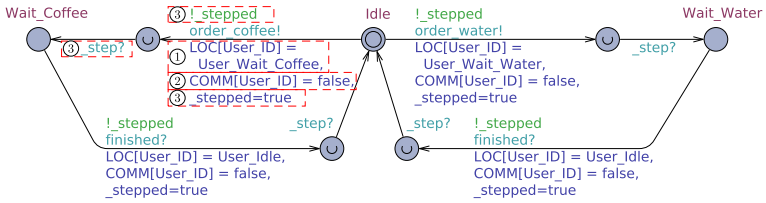
Figure 5 gives an initial impression of the results produced in this section, applied to the introductory example model in Fig. 1; it shows the resulting system obtained by inserting the most comprehensive version of the matcher template for the full set of supported observation traits, i.e., a system that supports matching of partial and potentially imprecise and delayed observations of variable data and (committed) locations. Figure 5a and b shows the adapted original model supporting tracking of active (1) and committed locations (2), and synchronization with the committed state matcher cycle (3). Figure 5c shows the inserted matcher template supporting initial delay (4) and matching of normal (5) and committed locations (6) with potential time and variable value deviation (7). Figure 5d shows the adapted code declarations introducing variables for storing the observable state and observation data, and functions for variable and commitment checking. Overall, Fig. 5 shows for the most advanced matching case that the complexity of the resulting system remains comparable with other tester systems used in literature (cf. [10, 16]). The steps toward the creation of such a matcher system are described in the following subsections.

### 4.1 Definition of matching

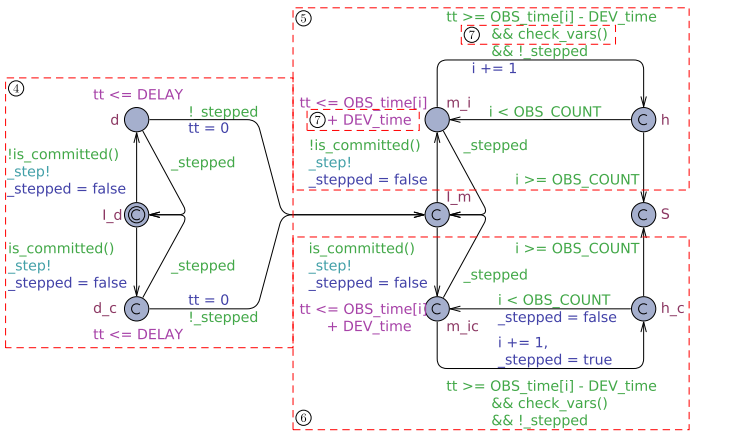
At first, we need to define the concept of *matching* in our context. We distinguish between matching of observation sequences (cf. Equation (7)), i.e., state-based matching, and observation traces (cf. Equation (9)), i.e., transition-based matching. Informally, we consider a given observation sequence (or observation trace) and a path (or trace) in the model as a match if the states (or actions) in the observation




(a) The resulting vending machine model




(b) The resulting user model




(c) The added observation test automaton

```
clock DM_ti, DM_ts, tt;
int i;
const int INST_COUNT = 2;
const int DM_ID = 0, User_ID = 1;
① const int UNNAMED_LOC = -1;
  const int DM_Error = 0, DM_Wait = 1, DM_Off = 2,
            DM_Make_Coffee = 3, DM_Make_Water = 4;
  const int User_Wait_Coffee = 0, User_Idle = 1,
            User_Wait_Water = 2;
  int LOC[INST_COUNT] = { DM_Off,User_Idle };
② int COMM[INST_COUNT] = { false,false };
  bool is_committed() {
    return exists (i : int[0, INST_COUNT - 1]) COMM[i];
  }
③ broadcast chan _step;
  bool _stepped = true;
④ const int DELAY = 3;
⑤ const int NOB = 0;
⑥ const int OBS_COUNT = 3;
  const int OBS_time[OBS_COUNT] = { 0,9,22 };
  const int HAS_OBS_time[OBS_COUNT] = { true,true,true };
  const int OBS_Drink_Machine[OBS_COUNT] =
    { DM_Off,DM_Make_Water,DM_Make_Coffee };
  const int HAS_OBS_Drink_Machine[OBS_COUNT] =
    { true,true,true };
  const int OBS_db[OBS_COUNT] = { 0,48,72 };
  const int HAS_OBS_db[OBS_COUNT] = { true,true,true };
  const int OBS_temp[OBS_COUNT] = { 1900,NOB,4100 };
  const int HAS_OBS_temp[OBS_COUNT] = { true,false,true };
⑦ const int DEV_time = 1;
  const int DEV_db = 2;
  const int DEV_temp = 100;
```

(d) The appended code declaration

**Fig. 5** The resulting matcher system for the example vending machine (Fig. 1) and observation sequence (Eq. (4)) from the introduction

are approximately modeled by the states of the model path (or actions of the model trace). *Approximately*, as opposed to *exact*, here means that expected imprecisions in measurements (e.g., due to limited sensor precision or differing starting states) are taken into account.

Formally, we define matching of an observation sequence $OBS_s = (obs_1, \ldots, obs_n)$ of variables $var_1, \ldots, var_k$ and a path $p = (s_1, \ldots, s_m)$ of a transition system $TS(M)$ of a timed automaton $M$ as the following relation, given an exact or approximate state matcher function $match_{state}$:

$$
\begin{aligned}
&match_{seq}(OBS_s, p, match_{state}) :=\\
&\exists(f : \{1,\dots,n\} \to \{1,\dots,m\}) :\\
&1.\, f(x) = y \implies match_{state}(obs_x, s_y) \text{(statematching)}\\
&2.\forall i \in \{1,\dots,n-1\} : f(i) \le f(i+1) \text{(monotonicity)}\\
&3.\, f(n) = m \text{ (suffix} - \text{free)} \qquad (10)
\end{aligned}
$$

In other words, each observation in $OBS_s$ (1) is matched (2) by exactly one state in order, and (3) the last observation is matched by the last state of path $p$ (omitting unnecessary suffixes).

The concrete state matcher function $match_{state}$ depends on the use case, and for our scope, we consider 4 different base matcher functions over time $t$ and variables $var_1, \dots, var_k$. These functions represent observation types encountered most often in real use cases due to partially non-observable system attributes, non-exact sensors, or different reference times of reality and model, and may also be combined. Thus, among the 4 types, $match_{state,eq}$ requires the equality of observation times and variables, $match_{state,part}$ allows matching of partial observations, $match_{state,dev}$ allows particular deviations of time and variable values, and $match_{state,delay}$ allows time shifts between the model states and the observations:

$$
\begin{aligned}
&match_{state,eq}(obs, s) := t_{obs} = u(t_g) \wedge\\
&\quad \forall i \in [1,k] : v_{obs}(var_i) = v(var_i) \qquad (11)\\
&match_{state,part}(obs, s) := t_{obs} = u(t_g) \wedge\\
&\quad \forall i \in [1,k] : v_{obs}(var_i) \in \{NOB, v(var_i)\}, \qquad (12)\\
&match_{state,dev}(dt, dvs)(obs, s) :=\\
&\quad |t_{obs} - u(t_g)| \le dt \wedge\\
&\quad \forall i \in [1,k] : |v_{obs}(var_i) - v(var_i)| \le dvs_i \qquad (13)\\
&match_{state,delay}(d)(obs, s) :=\\
&\quad t_{obs} + d = u(t_g) \wedge\\
&\quad \forall i \in [1,k] : v_{obs}(var_i) = v(var_i) \qquad (14)
\end{aligned}
$$

where $t_{obs}$ is the observation time, $v_{obs}(var_i)$ is the observed value of the variable $var_i$, $u(t_g)$ and $v(var_i)$ are the global time and variable valuations of the state $s = (l, u, v)$ of $TS(M)$, and $d \in \mathbb{R}_+$. The transition system $TS(M)$ matches the observation sequence $OBS_s$ if at least one such matching path $p$ exists.

Accordingly, we define matching of an observation trace $OBS_{tr} = (obs_1, \dots, obs_n)$ and an action trace $atr = (tn_1, \dots, tn_m)$ of a transition system $TS(M)$ of a timed automaton $M$ as follows, given an exact or approximate transition matcher function $match_{trans}$:

$$
\begin{aligned}
&match_{trace}(OBS_{tr}, atr, match_{trans}) :=\\
&\exists(f : \{1,\dots,n\} \to \{1,\dots,m\}) :\\
&1.\, f(x) = y \implies match_{trans}(obs_x, tn_y) \text{(trans.match.)}\\
&2.\forall i \in \{1,\dots,n-1\} : f(i) \le f(i+1) \text{ (monotonicity)}\\
&3.\, f(n) = m \text{ (suffix} - \text{free)} \qquad (15)
\end{aligned}
$$

Again, we define different matcher functions $match_{trans}$:

$$
\begin{aligned}
&match_{trans,eq}(obs, tn) :=\\
&\quad t_{obs} = t_{tn} \wedge a_{obs} = a_{tn} \qquad (16)\\
&match_{trans,dev}(dt)(obs, tn) :=\\
&\quad |t_{obs} - t_{tn}| \le dt \wedge a_{obs} = a_{tn} \qquad (17)\\
&match_{trans,delay}(d)(obs, tn) :=\\
&\quad t_{obs} + d = t_{tn} \wedge a_{obs} = a_{tn}, \qquad (18)
\end{aligned}
$$

where $t_{obs}$ is the observation time, $a_{obs}$ is an observed action, and $t_{tn}$ and $a_{tn}$ are the global trigger time and action of the transition $tn$. Here, the matcher functions only differ in handling of time, and no partial matching function exists for timed actions, as a timed action simply does not occur on an observation trace if it is not observed. Analogous to observation sequence matching, the transition system $TS(M)$ matches the observation trace $OBS_{tr}$ if at least one such matching action trace $atr$ exists.

### 4.2 The SRL logic

The matcher functions $match_{seq}$ and $match_{trace}$ describe the *sequential reachability* of observations, i.e., that observed states (or actions) are reached one after another on a path (or trace). Since we want to employ model checking to determine matching observation sequences and traces, we need to express sequential reachability as formal properties. For that purpose, we introduce $SRL$, the *sequence reachability logic*, and show that its formulae constitute duals of $SBLL$ formulae (introduced in [2] and recapitulated in Sect. A), which are already known to be testable (i.e., test automata are constructable for $SBLL$ properties). While the negation of sequential reachability properties can already be expressed in $SBLL$ (i.e., that all reachable paths never match the observation), and its result can be negated again, our helper logic $SRL$ provides a more natural way of describing sequential reachability (i.e., that a path exists that matches the observation). Using the dual relation between $SRL$ and $SBLL$ expressed in Lemma 2, we can translate the $SRL$ properties, which are not expressible in logics such as $TCTL$ commonly used for properties on $ETA$s, into $SBLL$ formulae, which can then be reduced into reachability problems via test automata (cf. [2]) and checked by model checkers like Uppaal.

In particular, the following 4 logical constructs are needed to express observation matching:

- Relational comparison of clocks and variables to integer bounds (e.g., $g = t \geq t_i$ or $g_{var} = v \geq v_i$) to express the check for equal or deviating time and data matches in individual states.
- Conjunctions of these relational comparisons for clocks ($g \wedge \varphi$) and variables ($g_{var} \wedge \varphi$).
- Existentiality checks over delays ($\exists\varphi$) to express the delays between observation points, i.e., to express that, from the initial or currently matched state, a path needs to exist on which after some delay the remaining observation points are matched.
- The existential freezing of clocks ($t \ \underline{in}_{\exists} \ \varphi$) to express the observation times relative to a (potentially initially delayed) "frozen" reference clock for time-shifted matching, i.e., to express that a path exists for which the complete observation matches relative to the reference clock.
- A check for enabled actions ($\langle a \rangle \varphi$) to express whether a particular action can be performed from a (matching) state.

Given these 4 constructs, the acceptance property ($\mathtt{tt}$), the usual logical conjunction ($\wedge$), and relational operations on clocks ($g$) and variables ($g_{var}$), the syntax of $SRL$ is defined as follows:

**Definition 1** ($SRL$ *syntax*) The $SRL$ syntax is inductively defined as:

$$\begin{array}{rcl} \varphi & ::= & \mathtt{tt} \mid g \wedge \varphi \mid (g_{var}) \wedge \varphi \mid \\ & & \exists\varphi \mid x \ \underline{in}_{\exists} \ \varphi \mid \langle a \rangle \varphi \\ g & ::= & x \sim q \mid x - y \sim q \\ g_{var} & ::= & \top \mid (x \sim_v q) \wedge g_{var} \end{array} \tag{19}$$

where $a \in \Sigma$, $x, y \in K$ (clocks), $q \in \mathbb{N}$, $\sim \in \{\leq, \geq\}$, and $\sim_v \in \{<, \leq, >, \geq, =, \neq\}$.

$\sim$ and $\sim_v$ are different sets of relational operators because the operators in $\sim$ are used to formulate clock invariants ($\leq$) and guards ($\geq$), while the operators in $\sim_v$ are used to formulate arbitrary constraints on variable values. The parentheses in variable constraints $(g_{var}) \wedge \varphi$ serve a particular purpose, as they *group* conjunctions of relational operations so that they are enforced at once; without such grouping, the individual constraints could potentially be applied on different states of $TS(M)$, making it impossible, e.g., to explicitly enforce variable values within an interval ($v \geq v_i \wedge v \leq v_i$).

**Definition 2** (*SRL semantics*) The $SRL$ semantics is inductively defined as the relation $\models$ on the transition system $TS(M)$ of a timed automaton $M$ with the extended states $(l, u, v)$, where $l$ is a location in $M$, $u$ is a clock valuation, and $v$ is a variable valuation:

$$\begin{array}{ll} & (l,u,v) \models \mathtt{tt} \\ \Leftrightarrow & true \\ & (l,u,v) \models g \wedge \varphi \\ \Leftrightarrow & g(u) \text{ and } \exists l', v' : (l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v') \\ & \quad \text{and } (l',u,v') \models \varphi \\ & (l,u,v) \models (g_{var}) \wedge \varphi \\ \Leftrightarrow & (g_{var})(v) \text{ and } \exists l', v' : (l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v') \\ & \quad \text{and } (l',u,v') \models \varphi \\ & (l,u,v) \models \exists\varphi \\ \Leftrightarrow & \exists d \in \mathbb{R}_{\geq 0} : \exists l', v' : (l,u,v) \xRightarrow{\varepsilon(d)} (l',u+d,v') \\ & \quad \text{and } (l',u+d,v') \models \varphi \\ & (l,u,v) \models \langle a \rangle \varphi \\ \Leftrightarrow & \exists l', v' : (l,u,v) \xRightarrow{a} (l',u,v') \\ & \quad \text{and } (l',u,v') \models \varphi \\ & (l,u,v) \models x \ \underline{in}_{\exists} \ \varphi \\ \Leftrightarrow & \exists l', v' : (l,u,v) \xrightarrow{\tau}{}^{*} (l',[x \to 0]u,v') \\ & \quad \text{and } (l',[x \to 0]u,v') \models \varphi \end{array}$$

***Example 3*** ($SRL$ property) The property that there exists a path on which the action $a$ can be triggered ($\langle a \rangle$) after 5 time units relative to a reference time $t_{ref}$, leading to the acceptance state ($\mathtt{tt}$) is expressed as $\varphi = t_{ref} \ \underline{in}_{\exists} \ \exists(t_{ref} = 5 \wedge \langle a \rangle \mathtt{tt})$. The corresponding observation sequence is $OBS := \langle (5, a) \rangle$.

Neither the variable state $v$ nor the checks via variable guards $g_{var}$ are available in $SBLL$, but $SBLL$ can be extended accordingly by adding the corresponding construct $(h_{var}) \vee \varphi$ with the following semantics:

$$\begin{array}{ll} & (l,u,v) \models (h_{var}) \vee \varphi \\ \Leftrightarrow & (h_{var})(v) \text{ or } \forall l', v' : (l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v') \\ & \quad \text{implies } (l',u,v') \models \varphi, \end{array} \tag{20}$$

with $h_{var} ::= \top \mid (x \sim_v p) \vee h_{var}$. Furthermore, the states of the remaining semantics of $SBLL$ need to be extended with the variable state $v$; however, the proofs in [2] for the testability of $SBLL$ properties still apply when adding $v$ to the states, as the semantic evaluation of the original $SBLL$ properties does not depend on $v$. For the newly added $(h_{var}) \vee \varphi$, the proof of $g \vee \varphi$ applies accordingly.

As stated earlier, $SRL$ properties are not expressible in the real-time logic $TCTL$, but they are expressible as test automata via the related logic $SBLL$. We define the translation from $SRL$ properties to $SBLL$ properties as a recursive function $srl\_to\_sbll : SRL \to SBLL$ as follows:

$$
\begin{aligned}
srl\_to\_sbll(\mathtt{tt}) &= \mathtt{ff} \\
srl\_to\_sbll(g \wedge \varphi) &= \neg g \vee srl\_to\_sbll(\varphi) \\
srl\_to\_sbll((g_{var}) \wedge \varphi) &= \neg(g_{var}) \vee srl\_to\_sbll(\varphi) \\
srl\_to\_sbll(\exists\varphi) &= \Cap srl\_to\_sbll(\varphi) \\
srl\_to\_sbll(x\ \underline{in}_{\exists}\ \varphi) &= x\ \underline{in}\ srl\_to\_sbll(\varphi) \\
srl\_to\_sbll(\langle a\rangle\varphi) &= [a]srl\_to\_sbll(\varphi), \qquad (21)
\end{aligned}
$$

For the translation function $srl\_to\_sbll$ and the relation between the properties of $SRL$ and $SBLL$, the following two lemmas hold:

**Lemma 1** ($srl\_to\_sbll$ returns $SBLL$ formulae) *Let $\varphi_{SRL}$ be a $SRL$ formula. Then, $\varphi_{SBLL} := srl\_to\_sbll(\varphi_{SRL})$ is a $SBLL$ formula.*

***Proof*** See section Appendix B.1. □

**Lemma 2** (Negative equivalence established by $srl\_to\_sbll$) *Let $TS(M)$ be a transition system of a timed automaton $M$, $\varphi$ be an $SRL$ property, and $srl\_to\_sbll(\varphi)$ be the corresponding $SBLL$ property. Then, $\neg(TS(M) \models_{SBLL} srl\_to\_sbll(\varphi)) \iff (TS(M) \models_{SRL} \varphi)$.*

***Proof*** See section Appendix B.2. □

### 4.3 Workflow for *SRL* checking

The workflow for observation matching as reachability problem can now be defined as follows:

1. Formulate the sequential reachability property $\varphi_{SRL}$ in $SRL$ for an observation $OBS$ and a timed automaton $M$.
2. Obtain the corresponding $SBLL$ property $\varphi_{SBLL}$ (cf. Lemma 2).
3. Compose a test automaton $T_{\varphi_{SBLL}}$ from $\neg\varphi$ based on predefined building blocks.
4. Construct the parallel composition $TS_{pc} = TS(M) \| TS(T_{\varphi_{SBLL}})$.
5. Check if $m_T$ is reachable on $TS_{pc}$, i.e., if $reach(TS_{pc}, m_T)$.
6. If so, conclude that $TS(M) \models \varphi_{SRL}$, or otherwise, $TS(M) \not\models \varphi_{SRL}$.

Step (1) uses our helper logic, $SRL$, to express matching of a given observation as formal property, i.e., that some path exists which matches the observation. Step (2) turns that $SRL$ property $\varphi_{SRL}$ into an $SBLL$ property $\varphi_{SBLL}$ (cf. Lemma 1 and Lemma 2) using the translation function $srl\_to\_sbll$ (Eq. (21)), for which the generation process of tester automata is known. The steps (3) to (6) then correspond to steps of the $SBLL$ checking workflow described in Sect. 3.3. However, the property $\varphi_{SBLL}$, by construction, expresses the negation of the original property $\varphi_{SRL}$, i.e., that on all paths, the observation data is not entirely matched; finding a counterexample path reaching $m_T$ in the test automaton $T_{\varphi_{SBLL}}$ is thus the desired outcome, meaning that $m_T$ acts as acceptance node rather than reject node as in $SBLL$ verification. Such a counterexample then represents a run that successfully matches the observation data. The following lemma holds for the workflow:

**Lemma 3** ($m_T$ reachable iff $M$ models $\varphi_{SRL}$) *Let $TS(M)$ be a transition system of a timed automaton $M$, $TS(T_{\varphi_{SBLL}})$ be a transition system of a test automaton $T_{\varphi_{SBLL}}$, $TS_{pc}$ be the parallel composition $TS(M) \| TS(T_{\varphi_{SBLL}})$, and $\varphi_{SRL}$ be an $SRL$ property. Then, $reach(TS_{pc}, m_T) \iff (TS(M) \models \varphi_{SRL})$.*

***Proof*** See Sect. B.3. □

### 4.4 Model synthesis

Given the helper logic $SRL$ introduced in Sect. 4.2, we now synthesize a basic model for observation matching, which we will use as base for further simplifications and extensions later in Sects. 4.5 and 4.6. For brevity, we write $t = t_i$ instead of $t \geq t_i \wedge t \leq t_i$ in $SRL$ formulae where suitable. In the most basic case, a sequence of full observations is matched exactly by a path (or trace) in the model, starting at the initial state of the model with a global clock $t$ initially set to 0. For matching an observation sequence $OBS = (t_1, v_1), \ldots, (t_n, v_n)$ of a single observable variable $v$, we get:

$$
\begin{aligned}
\varphi_{seq,SRL} = t\ \underline{in}_{\exists}\ \exists(t = t_1 \wedge (v = v_1) \wedge \exists(\ldots \\
\ldots \wedge \exists(t = t_n \wedge (v = v_n) \wedge \mathtt{tt}))), \qquad (22)
\end{aligned}
$$

where the final constraints $t = t_n \wedge (v = v_n)$ are satisfied in the acceptance state $\mathtt{tt}$. Likewise, for an observation trace $OBS = (t_1, a_1), \ldots, (t_n, a_n)$, we get the following $SRL$ formula:

$$
\begin{aligned}
\varphi_{tr,SRL} = t\ \underline{in}_{\exists}\ \exists(t = t_1 \wedge \langle a_1\rangle\exists(\ldots \\
\ldots \langle a_{n-1}\rangle\exists(t = t_n \wedge \langle a_n\rangle\ \mathtt{tt}))), \qquad (23)
\end{aligned}
$$

where $\langle a_n\rangle\mathtt{tt}$ expresses that the final action $a_n$ leads to the acceptance state $\mathtt{tt}$.

***Example 4*** (Exact state observation) For the vending machine (Fig. 1), we may observe exact states as shown in Eq. (1) in the introduction. That example observation sequence is expressed by the following SRL formula:

$$
\begin{aligned}
\varphi = t\ \underline{in}_{\exists}\ \exists(t = 0\ \wedge (db = 0 \wedge temp = 20) \\
\wedge \exists(t = 12\ \wedge (db = 50 \wedge temp = 20) \\
\wedge \exists(t = 25\ \wedge (db = 70 \wedge temp = 40) \wedge \mathtt{tt}))),
\end{aligned}
$$

and is matched, e.g., by the run shown in Eq. (3).

***Example 5*** (Exact action observation) For the vending machine (Fig. 1), we may observe exact actions as shown in Eq. (2) in the introduction. That example observation trace is expressed by the following SRL formula:

$$\varphi = t\ \underline{in}_{\exists}\ \exists(t = 11\ \wedge \langle \texttt{order\_water} \rangle \\ \exists(t = 14\ \wedge \langle \texttt{finished} \rangle \texttt{tt})),$$

and is matched, e.g., by the run shown in Eq. (3).

We need to show that the formulae $\varphi_{seq,SRL}$ and $\varphi_{tr,SRL}$ indeed correctly express observation matching in the sense of the matcher functions (cf. Section 4.1) $match_{seq}$ and $match_{trace}$ (based on $match_{state,eq}$ and $match_{trans,eq}$, respectively), i.e., that the following two lemmas hold:

**Lemma 4** ($\varphi_{seq,SRL}$ models observation sequence matching) *Let $TS(M)$ be a transition system of a timed automaton $M$, $\varphi_{seq,SRL}(OBS_{seq})$ be the SRL formula derived for the observation sequence $OBS_{seq}$, and $match_{seq}$ be the sequence matcher function. Then, $TS(M) \models \varphi_{seq,SRL}(OBS_{seq}) \iff \exists p \in P(M) : match_{seq}(OBS_{seq}, p, match_{state,eq})$.*

**Lemma 5** ($\varphi_{tr,SRL}$ models observation trace matching) *Let $TS(M)$ be a transition system of a timed automaton $M$, $\varphi_{tr,SRL}(OBS_{tr})$ be the SRL formula derived for the observation trace $OBS_{tr}$, and $match_{trace}$ be the trace matcher function. Then, $TS(M) \models \varphi_{tr,SRL}(OBS_{tr}) \iff \exists atr \in TR(M) : match_{trace}(OBS_{tr}, atr, match_{trans,eq})$.*

***Proof*** Given a timed automaton $M$ and an observation sequence $OBS_{seq}$, $TS(M) \models \varphi_{seq,SRL}(OBS_{seq})$ implies that—starting at $t = 0$—there exists a particular delay $d_1$ such that we reach the variable value $(t_1, v_1)$, followed by another delay $d_2$ reaching $(t_2, v_2)$, which is repeated for each $obs \in OBS_{seq}$ until the delay $d_n$ leads to $(t_n, v_n)$, matching the final observed variable value and leading to the acceptance state $\texttt{tt}$. That way, an accepting run traverses the path $p = \langle \ldots (t_1, v_1) \ldots (t_2, v_2) \ldots (t_n, v_n) \rangle$ (i.e., $p \in P(M)$) in $TS(M)$. For such a path, the reflexive relation $R = \{(obs_{seq,1} = (t_1, v_1), (t_1, v_1)), \ldots, (obs_{seq,n} = (t_n, v_n), (t_n, v_n))\}$ trivially fulfills the 4 criteria (state matching, function, order, and suffix-free) of $match_{seq}(OBS_{seq}, p, match_{state,eq})$ (cf. Equation (10)), and thus, Lemma 4 holds. The proof applies accordingly to Lemma 5 for $\varphi_{tr,SRL}(OBS_{tr})$ given an observation trace $OBS_{tr}$. □

Applying $srl\_to\_sbll$ (Eq. (21)) to the $SRL$ properties in Eqs. (22) and (23), we get the following $SBLL$ properties, where $t_i < t \vee t_i > t$ negates the $SRL$ constraint $t_i = t$ (cf. Section A for the recapitulation of the $SBLL$ syntax and semantics):

$$\begin{aligned} \varphi_{seq,SBLL} &= srl\_to\_sbll(\varphi_{seq,SRL}) \\ &= t\ \underline{in}\ \Cap(t < t_1 \vee t > t_1 \vee \\ &\quad (v < v_1 \vee v > v_1) \vee \Cap(\ldots \\ &\quad \ldots \vee \Cap(t < t_n \vee t > t_n \vee \\ &\quad (v < v_n \vee v > v_n) \vee \texttt{ff}))) \end{aligned} \tag{24}$$

$$\begin{aligned} \varphi_{tr,SBLL} &= srl\_to\_sbll(\varphi_{tr,SRL}) \\ &= t\ \underline{in}\ \Cap(t < t_1 \vee t > t_1 \vee [a_1]\Cap(\ldots \\ &\quad \ldots [a_{n-1}]\Cap(t < t_n \vee t > t_n \vee [a_n]\texttt{ff}))) \end{aligned} \tag{25}$$

The resulting test automata, which are composed from the building blocks introduced in [2] (cf. Section 3.3), are shown in Fig. 6, and implement the following steps: Initially, the edge $m_0 \to m_{1,1}$ (implementing the freeze operator $t\ \underline{in}\ \varphi$) introduces a clock $t := 0$ at global time 0, which is used as a reference time for the observation data. Afterward, a sequence of 5 locations $(m_{i,1}, \ldots, m_{i,5})$ is repeated for each individual observation data point $obs_i, 1 \leq i \leq |OBS|$. In each of these sequences, first, $m_{i,1}$ allows an arbitrary delay (implementing $\Cap\varphi$). Then, the edges $m_{i,2} \to m_{i,3}$ and $m_{i,3} \to m_{i,4}$ perform the check against the time stamp of the observation (implementing $t < t_i \vee t > t_i$ of the $SBLL$ formula). The next part is different for observation sequences and traces: For observation sequences, the edges $m_{i,4} \to m_{i,5}$ and $m_{i,5} \to m_{i+1,1}$ check whether the observed variable value is exactly matched (implementing $v < v_i \vee v > v_i$ of the $SBLL$ formula); in the model, we write $v = v_i$ instead of $v \geq v_i \wedge v \leq v_i$ for brevity where suitable. For observation traces, these edges check whether the observed timed action can be performed (implementing $[a_i]$). In both cases, the latter edge leads to the start of the location sequence for the next observation. The last location of the last location sequence (i.e., $m_{n,5}$) eventually leads to the final location $m_T$.

For the derived test automata $T$, we need to prove that the following lemma holds:

**Lemma 6** ($T$ correctly models observation matching) *Let $OBS$ be an observation sequence or trace. Then, the test automaton $T$ derived from $OBS$ correctly models observation matching in the sense of the matcher functions in Eqs. (10) and (15).*

***Proof*** We know from Lemma 4 and Lemma 5 that the formulae $\varphi_{SRL}$ correctly model observation matching as defined in Eqs. (10) and (15), respectively, from Lemma 1 and Lemma 2 that the subsequent translation from $SRL$ to $SBLL$ is correct, and from Lemma 3 and [2] that the synthesis of the test automaton is correct. Thus, Lemma 6 holds. □

***Remark 1*** (Match actions as variables) Up to this point, we handled the actions in observation traces similar to [2], where

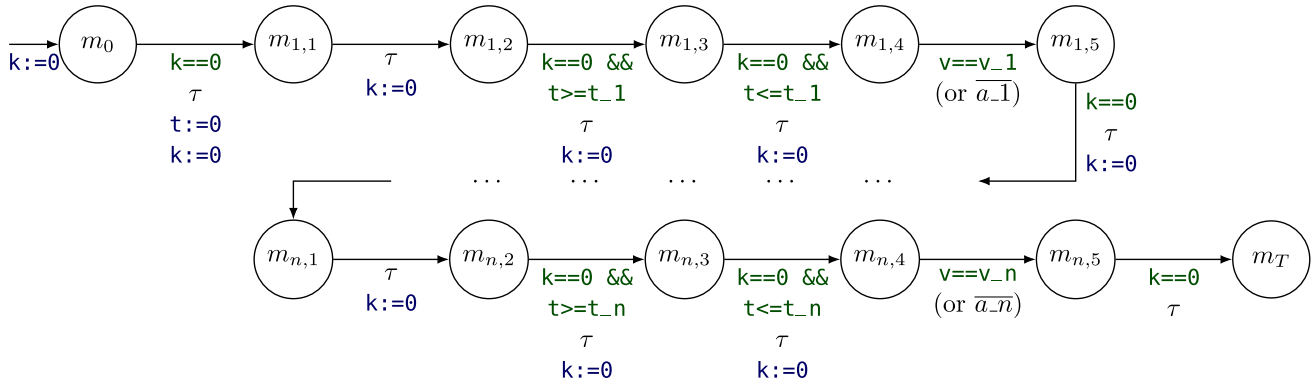



**Fig. 6** The resulting test automaton for the properties $\varphi_{seq}$ and $\varphi_{tr}$ (the latter differs by the parts in parentheses)

the test automaton synchronizes with the original model via such actions (calling `a!` when then model calls `a?`, and vice versa), and thus, becomes an explicit actor in the composed system. In our case, however, we want the test automaton to act as a mere observer, i.e., to only check if the original model components synchronized on a particular action without partaking in the action itself. We make synchronizations observable by introducing a data variable `ACT` $\in$ {`None`, `a_1`, ..., `a_n`}, which is assigned `None` or `a_i` when no action or $a_i$ is observed, respectively. The edges of the original model are split into two edges over urgent intermediate locations (similar to Fig. 2 without `sync2` and with urgency instead of commitment), where `ACT` is assigned the action `a_i` on the first edge, and again assigned `None` on the second edge. While the original model is in an intermediate location, where no additional time can pass due to the urgency, the test automaton can check for the value of `ACT` at the exact time of the triggered action.

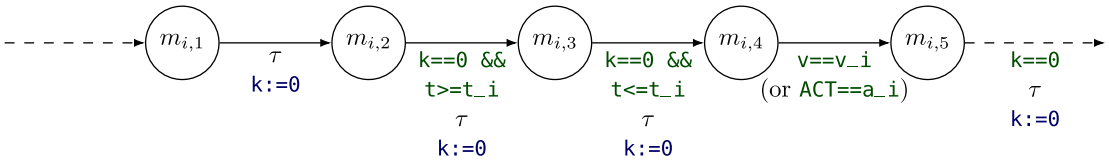
Observing the original model in such manner has two advantages: First, it allows the matching of observations in system models that are "self-contained" (i.e., where the interacting entities are all contained in the model and no input is required from the outside), as binary synchronizations in particular do not allow a test automaton to participate as a third party. Second, handling actions as variables allows formulating more general templates applicable to both observation traces and observation sequences, which we will do for the remainder of this article.

## 4.5 Model simplification

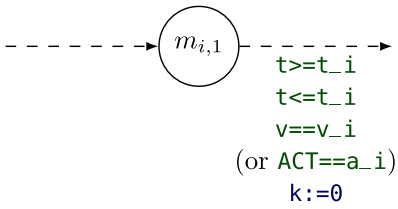
The base version of the test automaton requires $|OBS|*5+2$ locations, i.e., the number of locations grows linearly with the number of observations. Furthermore, its state space is more extensive than required for observation matching, as the clock constraints (implemented as guards) allow infinite progress in the original model processes even if an observation cannot be matched anymore. Via a set of simplifications, we can obtain a test automaton that is more compact in terms of model size (requiring only 4 locations, independent of the number of observations) and state space size (omitting non-matching paths early on), and for which the extensions presented in Sect. 4.6 are easier to implement. Note that the behavior of the base and simplified model versions are only required to be equivalent regarding the reachability of $m_T$, which we will prove at the end of this section.

The location sequence for a single observation data point (Fig. 7a) can be simplified in 3 steps: First, we can infer that $k = 0$ for clock $k$ also needs to hold on the edge $m_{i,4} \rightarrow m_{i,5}$, as $k$ is not reset on $m_{i,4} \rightarrow m_{i,5}$ and $k = 0$ is required on the following edge $m_{i,5} \rightarrow$; from $k = 0$ on all edges we can conclude that no time passes from $m_{i,2}$ to $m_{i,5}$. Thus, second, we can merge all edges into a single edge labeled with both the constraints on $t$ ($t \geq t_i$ and $t \leq t_i$) and $v$ (Fig. 7b). Third, we can turn the upper bound guard $t \leq t_i$ into an invariant on $m_{i,1}$ (Fig. 7c), so that, instead of disabling the edge $m_{i,1} \rightarrow$ (when $v_i$ was not assigned to $v$ while $t \leq t_i$), the path runs into a deadlock. While both approaches make $m_T$ unreachable once $t > t_i$, the invariant-based version allows the model checker to omit the path early on, while the guard-based version still allows infinite time progression in $m_{i,1}$, and thus, further steps in the original model processes can occur, even though $OBS$ is then already known to be non-matching.

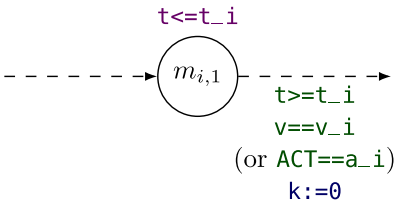
Applying the described simplifications to the location sequences of each observation data point, we obtain a test automaton with $|OBS| + 2$ locations as shown in Fig. 8a. To finally remove the linear dependency between the number of locations and the size of $OBS$, we can make explicit use of indexing instead, i.e., use array structures as supported by Uppaal and increment the index for accessing these arrays on each loop (Fig. 8b). In that final solution, we introduce the committed helper location $h$ for the decision whether more data points exist after a successful match ($i < |OBS|$), and after turning $m_0$ into an urgent location (Fig. 8c) by omitting




(a) The test automaton section for a single observation data point (cf. Fig. 6)

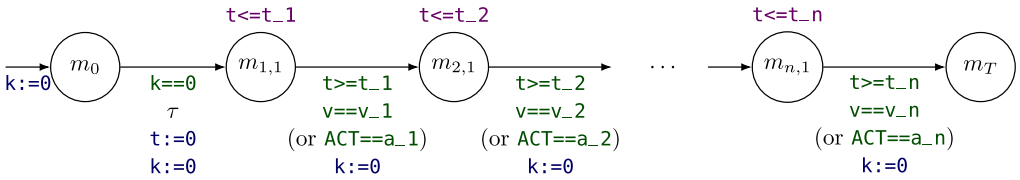



(b) Merge locations

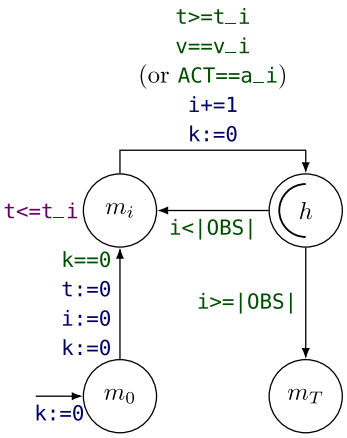



(c) Transform guard to invariant

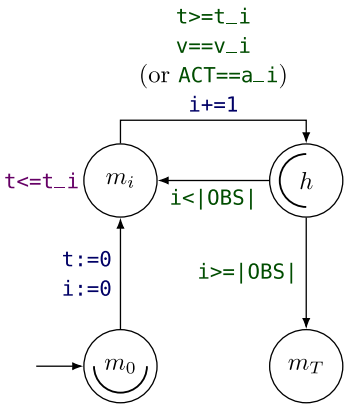
**Fig. 7** The simplification steps of the test automaton section for a single observation data point




(a) The test automaton (cf. Fig. 6) with simplified sections (cf. Fig. 7)




(b) Access observation data points via array indexing (with committed helper location $h$)




(c) Remove unnecessary clock $k$ (making $m_0$ urgent)

**Fig. 8** The simplification steps of the test automaton

$k := 0$ and $k = 0$ (cf. Section 3.1.2), remove the clock $k$ completely, as no constraints on $k$ occur anymore.

To prove that the simplified test automaton $T_s$ preserves the matcher property of the original test automaton $T$, we have to show that the following lemma regarding the reachability of $m_T$ holds:

**Lemma 7** (Equivalence of $T$ and $T_s$ regarding reachability of $m_T$) *Let $TS(M)$ be a transition system of a timed automaton $M$, and $TS(T)$ and $TS(T_s)$ be the transition systems of the original and simplified test automata $T$ and $T_s$, respectively. Then, $reach(TS(M) \| TS(T), m_T) \iff reach(TS(M) \| TS(T_s), m_T)$*

***Proof*** In the parallel composition of $TS(M)$ and $TS(T)$ of the original test automaton $T$ (Fig. 6), the acceptance node $m_T$ is only reached if the location sections $m_{i,1}$ to $m_{i+1,1}$ are successfully traversed for all $i \in [1, n]$. While the number of possible concrete runs through such location sections is infinite, each run has the following general form, consisting of potentially multiple transitions in $TS(M)$ in between each transition in $TS(T)$.

$$
\begin{aligned}
&(s_0 \| \langle m_{i,1}, u_0, v_0 \rangle) && \xRightarrow{\varepsilon(d)} (s_1 \| \langle m_{i,1}, u_1, v_1 \rangle) \\
\xrightarrow{\tau} &(s_1 \| \langle m_{i,2}, u_2, v_2 \rangle) && \xrightarrow{\tau}{}^{*} (s_2 \| \langle m_{i,2}, u_3, v_3 \rangle) \\
\xrightarrow{\tau} &(s_2 \| \langle m_{i,3}, u_4, v_4 \rangle) && \xrightarrow{\tau}{}^{*} (s_3 \| \langle m_{i,3}, u_5, v_5 \rangle) \\
\xrightarrow{\tau} &(s_3 \| \langle m_{i,4}, u_6, v_6 \rangle) && \xrightarrow{\tau}{}^{*} (s_4 \| \langle m_{i,4}, u_7, v_7 \rangle) \\
\xrightarrow{\tau} &(s_5 \| \langle m_{i,5}, u_8, v_8 \rangle) && \xrightarrow{\tau}{}^{*} (s_6 \| \langle m_{i,5}, u_9, v_9 \rangle) \\
\xrightarrow{\tau} &(s_6 \| \langle m_{i+1,1}, u_{10}, v_{10} \rangle),
\end{aligned}
$$

where each $s$ is a state of $TS(M)$, and each $\langle m, u, v \rangle$ is a state of $TS(T)$, where $m$ is the active location, and $u$ and $v$ are concrete valuation of clocks and variables, respectively. For the individual transitions allowed by each test automaton component which such a location section is composed of, see the proof provided in [2] on the correct testing for $SBLL$. As no time passes except for the initial delaying computation $\xRightarrow{\varepsilon(d)}$, we know that all clock valuations apart from $u_0$ are equal, i.e., $u_i = u_j$ for all $i, j \in [1, 10]$. Furthermore, we know that $(t \geq t_i)(u_4)$ and $(t \leq t_i)(u_6)$ hold due to the edge guards, and thus, due to $u_4 = u_6 = u_{10}$, we know for the final clock valuation $u_{10}$ of the location section that $(t \geq t_i)(u_{10})$ and $(t \leq t_i)(u_{10})$ hold. Thus, $(s_6 \| \langle m_{i+1,1}, u_{10}, v_{10} \rangle)$ can be any state with $(t = t_i)(u_{10})$ reachable from $(s_0 \| \langle m_{i,1}, u_0, v_0 \rangle)$ with the variable valuation $v_i$.

In the parallel composition of $TS(M)$ and $TS(T_s)$ of the simplified test automaton $T_s$ (Fig. 8c), the acceptance node $m_T$ is only reached if the location sections $m_{i,1}$ are successfully traversed for all $i \in [1, n]$. Here, each run has the following general form:

$$
\begin{aligned}
&(s_0 \| \langle m_{i,1}, u_0, v_0 \rangle) && \xRightarrow{\varepsilon(d)} (s_1 \| \langle m_{i,1}, u_1, v_1 \rangle) \\
\xrightarrow{\tau} &(s_1 \| \langle m_{i+1,1}, u_2, v_2 \rangle)
\end{aligned}
$$

We know that $u_1 = u_2$. Furthermore, $(t \leq t_i)(u_1)$ holds due to the location invariant in $m_{i,1}$, and $(t \geq t_i)(u_2)$ holds due to the edge guard. Thus, it follows that $(s_1 \| \langle m_{i+1,1}, u_2, v_2 \rangle)$—as for the original test automaton—can be any state with $(t \geq t_i \wedge t \leq t_i)(u_2)$ reachable from $(s_0 \| \langle m_{i,1}, u_0, v_0 \rangle)$ with the variable valuation $v_i$. We conclude that, even though the simplified automaton has fewer states in total (i.e., as infinite stays in locations on already non-matching runs are prevented now), the set of states reachable in $(M \| T)$ and $(M \| T_s)$ are equal after each location section. Finally, as the introduction of explicit indexing and the committed helper location $h$ (Fig. 8b) do not alter reachability of $m_T$, we conclude that Lemma 7 holds. □

## 4.6 Model extension

In Sects. 4.4 and 4.5, we derived a base test automaton that matches full and exact action and variable observations in a setting where the modeled and observed clocks are synchronized and only normal and urgent intermediate states are observable. Real observations, however, often go along with various types of imprecisions, and the use case may further dictate whether measurements are performed qualitatively (i.e., a specific location is visited) or rather quantitatively. In this section, we will extend the matcher system for matching of:

- partial observations,
- deviating observations,
- location observations,
- time-shifted observations, and
- observations in committed locations.

Table 1 gives an overview of the observation types. The first two observation types (basic trace, basic sequence) are the base types that we already introduced in Sect. 4.4. We categorize the remaining observation types as follows: First, we distinguish between those observations that are expressible by SRL properties (partial, deviating, locations, time shift) and those that require manual adaption of the matcher system (committed locations). Second, among the observation types expressible in SRL, we distinguish between properties where the sections for individual data points are affected (partial, deviating, locations) and those where the existential operators outside of the data point sections are affected (time shift) (Table 1).

We describe the observation types and their corresponding formulae and models in the following, and motivate the observation types by examples based on the introductory model in Fig. 1.

### 4.6.1 Observation of partial, deviating, and location data

Matching of partial, deviating, and location data is achieved by adapting the data point sections of the basic SRL property. The resulting SRL properties are shown in the rows 3–5 of Table 1. In the case of partial observations, we use $NOB$ to indicate that a particular variable $v_i$ was not observed in a data point; as result, the SRL property contains $NOB$ instead of $v_i = val_i$, where $NOB$ always evaluates to $true$ independent of the actual variable value. In the case of deviation, we provide the maximal allowed deviations $dt, dv_1, \ldots, dv_n$, and the SRL formula sections $t = t_i$ and $v_i = val_i$ become $t \geq t_i - dt \wedge t \leq t_i + dt$ and $v_i \geq val_i - dv_i \wedge v_i \leq val_i + dv_i$, respectively. Finally, for matching of a location $l$ of a timed automaton $M$, we add $LOC[M] = l$ to the data point.

Note for matching of partial observations that one may consider to simply remove $v_i$ from formula sections, however, that approach would not be compatible with explicit indexing as used in our simplified test automaton. Further-

**Table 1** The observation data and corresponding SRL properties for different observation types

| Observation type | Observation data $OBS$ | SRL Property $\varphi$ |
|---|---|---|
| Basic sequence | $(t_1, val_{1,1}, \ldots, val_{n,1}), \ldots$ $(t_m, val_{1,m}, \ldots, val_{n,m})$ | $\varphi_{seq,SRL} = t\ \underline{in}_\exists\ \exists(t = t_1 \wedge \langle v_1 = val_{1,1} \wedge \ldots \ldots \wedge v_n = val_{n,1}\rangle \wedge \exists(\ldots \ldots \wedge \exists(t = t_m \wedge \langle v_1 = val_{1,m} \wedge \ldots \ldots \wedge v_n = val_{n,m}\rangle \wedge \mathtt{tt})))$ |
| Basic trace | $(t_1, a_1), \ldots, (t_m, a_m)$ | $\varphi_{tr,SRL} = t\ \underline{in}_\exists\ \exists(t = t_1 \wedge \langle a_1\rangle\exists(\ldots \ldots \langle a_{m-1}\rangle\exists((t = t_m \wedge \langle a_m\rangle \mathtt{tt})))$ |
| Partial data | $(t_1, \boxed{NOB}, \ldots, val_{n,1}), \ldots$ $(t_m, val_{1,m}, \ldots, \boxed{NOB})$ | $\varphi_{part,SRL} = t\ \underline{in}_\exists\ \exists(t = t_1 \wedge \langle \boxed{v_1 = NOB} \wedge \ldots \ldots \wedge v_n = val_{n,1}\rangle \wedge \exists(\ldots \ldots \wedge \exists((t = t_m \wedge \langle v_1 = val_{1,m} \wedge \ldots \ldots \wedge \boxed{v_n = NOB}\rangle \wedge \mathtt{tt})))$ |
| Deviation | Any sequence (e.g., basic), deviations $\boxed{dt, dv_1, \ldots, dv_n}$ | $\varphi_{dev,SRL} = t\ \underline{in}_\exists\ \exists(t \geq \boxed{t_1 - dt} \wedge t \leq \boxed{t_1 + dt} \wedge \langle \ldots \wedge v_n \geq \boxed{val_{n,1} - dv_n} \wedge v_n \leq \boxed{val_{n,1} + dv_n}\rangle \wedge \exists(\ldots \wedge \mathtt{tt}))$ |
| Location data | $(t_1, \boxed{l_1}), \ldots, (t_m, \boxed{l_m})$ | $\varphi_{loc,SRL} = t\ \underline{in}_\exists\ \exists(t = t_1 \wedge (\boxed{LOC[M] = l_1}) \wedge \exists(\ldots \ldots \wedge \exists(t = t_m \wedge (\boxed{LOC[M] = l_m}) \wedge \mathtt{tt})))$ |
| Time shift | Any (e.g., basic sequence), maximum delay $\boxed{D}$ | $\varphi_{delay,SRL} = \boxed{t_d\ \underline{in}_\exists\ \exists(t_d \leq D} \wedge t\ \underline{in}_\exists\ \exists(t = t_1 \wedge (v_1 = val_{1,1} \wedge \ldots) \wedge \exists(\ldots \ldots \wedge \exists(t = t_m \wedge \langle v_1 = val_{1,m} \wedge \ldots\rangle \wedge \mathtt{tt}))))$ |
| Committed | Any | Not expressed as property |

more, location matching may require the addition of helper variables (see Fig. 5) that track the currently active location.

For the formulae $\varphi_{part,SRL}$, $\varphi_{dev,SRL}$, and $\varphi_{loc,SRL}$, the following lemma holds:

**Lemma 8** ($\varphi_{part,SRL}$, $\varphi_{dev,SRL}$, and $\varphi_{loc,SRL}$ model partial, deviating, and location matching) *Let* $type \in \{part, dev, loc\}$ *be the matcher type,* $TS(M)$ *be a transition system of a timed automaton* $M$, $\varphi_{type,SRL}(OBS_{seq,type})$ *be the SRL formula derived for the observation sequence* $OBS_{seq,type}$ *(cf. Table 1), and* $match_{seq}$ *be the sequence matcher function (cf. Equation (10)). Then,* $TS(M) \models \varphi_{type,SRL}(OBS_{seq,type}) \iff \exists p \in P(M) : match_{seq}(OBS_{seq,type}, p, match_{state,type})$,

i.e., the formulae correctly model the matching of partial, deviating, and location observations.

***Proof*** See Sect. B.4. □

***Example 6*** (Partial observations) For the vending machine (Fig. 1), measuring the temperature may not be relevant (and thus not performed) while water is prepared, so that no temperature data are given for that time. The example observation $\langle(6\,s, 0\,db, 20\,°\mathrm{C}), (12\,s, 50\,db, NOB), (25\,s, 70\,db, 40\,°\mathrm{C})\rangle$ is expressed by the following SRL formula:

$$\varphi = t\ \underline{in}_\exists\ \exists(t = 6\ \wedge (db = 0 \wedge temp = 20) \wedge \exists(t = 12\ \wedge (db = 50 \wedge temp = NOB) \wedge \exists(t = 25\ \wedge (db = 70 \wedge temp = 40) \wedge \mathtt{tt}))),$$

and is matched, e.g., by the following (simplified) run:

$$\begin{aligned} &\langle \mathtt{Off}, \{(\mathtt{t}, 0)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Wait}, \{(\mathtt{t}, 6)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Make\_Water}, \{(\mathtt{t}, 12)\}, \{(\mathtt{db}, 50), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Wait}, \{(\mathtt{t}, 15)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Make\_Coffee}, \{(\mathtt{t}, 25)\}, \{(\mathtt{db}, 70), (\mathtt{temp}, 40)\}\rangle \end{aligned}$$

***Example 7*** (Deviating observations) For the vending machine (Fig. 1), the temperature and sound levels measured via sensors are never exact due to measurement uncertainty and the limited precision of stored data. The example observation $\langle(6\,s, 0\,db, 19\,°\mathrm{C}), (14\,s, 68\,db, 31\,°\mathrm{C})\rangle$ is expressed by the following SRL formula with allowed deviations `dev_db=2db` and `dev_temp=1°C`:

$$\varphi = t\ \underline{in}_\exists\ \exists(t = 6 \wedge (db \geq -2 \wedge db \leq 2 \wedge temp \geq 19 \wedge temp \leq 21) \wedge \exists(t = 14 \wedge$$

$$(db \geq 68 \wedge db \leq 72 \wedge temp \geq 29 \wedge temp \leq 31) \wedge \mathtt{tt})),$$

and is matched, e.g., by the following (simplified) run:

$$\begin{aligned} &\langle \mathtt{Off}, \{(\mathtt{t}, 0)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Wait}, \{(\mathtt{t}, 6)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Make\_Coffee}, \{(\mathtt{t}, 14)\}, \{(\mathtt{db}, 70), (\mathtt{temp}, 30)\}\rangle \end{aligned}$$

***Example 8*** (Location observations) For the vending machine (Fig. 1), we may not want to measure the actual temperature and sound levels via sensors, but only qualitatively observe if the machine is currently waiting for input, or preparing one of the two beverages. The example observation $\langle (6\,s, \mathit{Wait}), (18\,s, \mathit{Make\_Coffee}), (25\,s, \mathit{Make\_Water})\rangle$ is expressed by the following SRL formula:

$$\begin{aligned} \varphi = t\ \underline{in}_{\exists}\ \exists(t = 6\ \wedge (LOC[M] = \mathit{Wait}) \\ \wedge \exists(t = 18\ \wedge (LOC[M] = \mathit{Make\_Coffee}) \\ \wedge \exists(t = 25\ \wedge (LOC[M] = \mathit{Make\_Water}) \wedge \mathtt{tt}))), \end{aligned}$$

and is matched, e.g., by the following (simplified) run:

$$\begin{aligned} &\langle \mathtt{Off}, \{(\mathtt{t}, 0)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Wait}, \{(\mathtt{t}, 6)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Make\_Coffee}, \{(\mathtt{t}, 18)\}, \{(\mathtt{db}, 70), (\mathtt{temp}, 50)\}\rangle \\ \rightarrow\ &\langle \mathtt{Wait}, \{(\mathtt{t}, 22)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 35)\}\rangle \\ \rightarrow\ &\langle \mathtt{Make\_Water}, \{(\mathtt{t}, 25)\}, \{(\mathtt{db}, 50), (\mathtt{temp}, 20)\}\rangle \end{aligned}$$

#### 4.6.2 Observation of time-shifted data

The case of matching time-shifted data is shown in row 6 of Table 1 and is also expressible as SRL property. In the adapted SRL property, we prepend "$t_d\ \underline{in}_{\exists}\ \exists t_d \leq D \wedge \ldots$" to the base property, which introduces a clock $t_d$ that allows a delay of $d \leq D$ time units before switching into the actual matching section; as result, the clock $t$ of the observations is now shifted by $d$ time units compared to the global clock of the model.

For the formula $\varphi_{delay,SRL}$, the following lemma holds:

**Lemma 9** ($\varphi_{delay,SRL}$ models time-shifted observation matching) *Let $TS(M)$ be a transition system of a timed automaton $M$, $\varphi_{delay,SRL}(OBS_{seq,delay})$ be the SRL formula $\varphi_{delay,SRL}$ (cf. Table 1) derived for the observation sequence $OBS_{seq,delay}$, and $match_{seq}$ be the sequence matcher function (cf. Equation (10)). Then, $TS(M) \models \varphi_{delay,SRL}(OBS_{seq,delay}) \iff \exists p \in P(M) : match_{seq}(OBS_{seq,delay}, p, match_{state,delay})$,*

i.e., the formula correctly models the matching of time-shifted observations

***Proof*** See Sect. B.5. □

***Example 9*** (Time-shifted observations) For the vending machine (Fig. 1), we may begin the observation not when the machine is still turned off (i.e., the initial state of the model), but already activated and waiting for instructions for some time. The example observation $\langle (0\,s, \mathit{Wait}), (3\,s, \mathit{Make\_Coffee})\rangle$ is expressed by the following SRL formula with a delay $d = 10$:

$$\begin{aligned} \varphi = t_d\ \underline{in}_{\exists}\ \exists(t_d \leq 10 \wedge t\ \underline{in}_{\exists} \\ \exists(t = 0\ \wedge (LOC[M] = \mathit{Wait}) \\ \wedge \exists(t = 3\ \wedge (LOC[M] = \mathit{Make\_Coffee}) \wedge \mathtt{tt}))), \end{aligned}$$

and is matched, e.g., by the following (simplified) run:

$$\begin{aligned} &\langle \mathtt{Off}, \{(\mathtt{t}, 0)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Wait}, \{(\mathtt{t}, 10)\}, \{(\mathtt{db}, 0), (\mathtt{temp}, 20)\}\rangle \\ \rightarrow\ &\langle \mathtt{Make\_Coffee}, \{(\mathtt{t}, 13)\}, \{(\mathtt{db}, 70), (\mathtt{temp}, 25)\}\rangle \end{aligned}$$

***Example 10*** (Combined data imprecisions) For the vending machine (Fig. 1), we may observe combinations of imprecisions, as shown in the observation Eq. (4) presented in the introduction. Given the previously introduced $SRL$ constructs, that example observation is expressed by the following SRL formula with allowed deviations `dev_db=2db` and `dev_temp=1°C`, and with a delay $d = 3$:

$$\begin{aligned} \varphi = t_d\ \underline{in}_{\exists}\ \exists(t_d \leq 3 \wedge t\ \underline{in}_{\exists}\ \exists(t = 0 \wedge \\ (db \geq -2 \wedge db \leq 2 \wedge temp \geq 19 \wedge temp \leq 21) \\ \wedge \exists(t = 9 \wedge \\ (db \geq 48 \wedge db \leq 52 \wedge temp = NOB) \\ \wedge \exists(t = 22 \wedge \\ (db \geq 68 \wedge db \leq 72 \wedge temp \geq 39 \wedge temp \leq 41) \\ \wedge \mathtt{tt})))). \end{aligned}$$

That query evaluates to true on the example model and thus decides that the observation is sufficiently modeled under consideration of imprecision, i.e., that the observation points still lie in defined regions around existing model states. The observation is matched, e.g., by the run shown in Eq. (3).

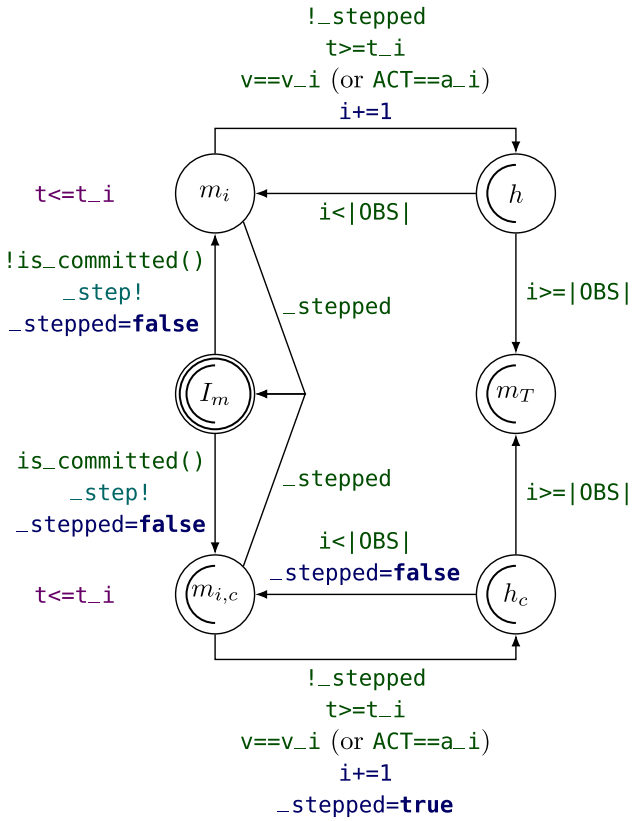
#### 4.6.3 Observation of committed locations

The base matcher system does not allow matching against data of committed locations, as these states are left immediately without the possibility to traverse the matcher cycle, i.e., the cycle over $m_i$ and $h$. In contrast to the other extensions, matching of committed locations cannot be achieved by adapting the SRL property, as it concerns the order of transitions (i.e., committed transitions before normal ones)

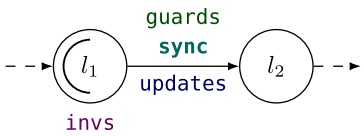
rather than their enabling (i.e., only transitions that satisfy the constraints imposed by the observation). Therefore, the matcher system requires the introduction of an artificial synchronization between the test automaton and the original model, which enables a matcher check before the committed location needs to be left. The solution shown in Fig. 9 applies the following 4 changes to the base matcher system:

1. A copy of the *matcher cycle* (via locations $m_{i,c}$ and $h_c$) for checking of committed locations is added (see lower half of Fig. 9a). In contrast to the base matcher cycle (via locations $m_i$ and $h$), $m_{i,c}$ is a committed location, which allows checking observation conditions before the original model is forced to leave its committed location.
2. Synchronization via the variable `_stepped` and the broadcast channel `_step` is added, as $m_{i,c}$ and the committed location in the original model need to become active simultaneously. The variable `_stepped` has 2 purposes: First, it indicates that a transition in the original model was initiated, blocking any further transitions in the original model and matchings in the test automaton (via guard `!_stepped`) until synchronization with the test automaton was finished. Second, it indicates that an observation data point was matched from $m_{i,c}$, blocking transitions in the original model until we return to $m_{i,c}$. The channel `_step` forces the original model to take its transitions simultaneously to the transitions into the next checking location ($m_i$ or $m_{i,c}$) in the test automaton. We call the cycle over the locations $m_i$ (or $m_{i,c}$) and $I_m$ the *stepper cycle*.
3. The edges in the original model (Fig. 9b) are split into two edges over an intermediate location (Fig. 9c), as the additional synchronization via `_step` may result in two synchronizations on a single edge (cf. Section 3.1.2). Note that in contrast to the approach described in Sect. 3.1.2, we define the intermediate location $l_{int}$ as urgent instead of committed, as otherwise, the intermediate location would need to be left before the test automaton can take the edge $m_{i,c} \rightarrow I_m$, resulting in a deadlock. However, as further transitions are blocked by `_stepped` anyway, the effect of $l_{int}$ remains the same.
4. A function `is_committed()` is added, which checks if a committed location is active in the following state of the original model. A helper variable indicating whether the current location is committed is added to each original model process, and its value is updated on the incoming edges of each location (see *COMM[id]=false* in Fig. 9c).

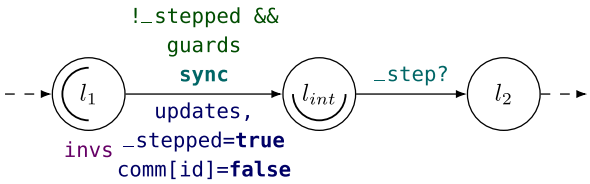
If initial delay (cf. Section 4.6.2) is allowed, the delay section of the model needs to be adapted accordingly (see Fig. 10), as we want to be able to switch from the delay section to the matcher section even if the original model is currently in a committed location.




(a) The adapted test automaton




(b) A section of an original model




(c) The model section adapted for matcher synchronization

**Fig. 9** The original model and test automaton supporting committed location matching

***Remark 2*** (Match actions in the committed test automaton) In Remark 1, we described how the matching of actions can be implemented as variable matching. Action matching works similarly in the committed test automaton; note, however, that the check for `!_stepped` on the edge $m_i \rightarrow h$ needs to be replaced by `_stepped` then, as the action check should in fact be performed after a step into the intermediate location $l_{int}$ was performed in the original model.

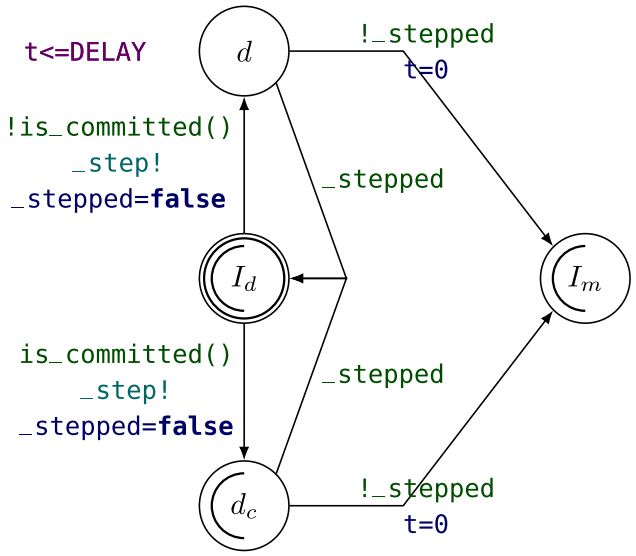



**Fig. 10** The delay section supporting the switch into the matching section in committed states

***Example 11*** (Committed locations) For the vending machine (Fig. 1), we may explicitly observe when the machine shuts down due to overheating, i.e., when we reach the committed `Error` location. The example observation $(6\,s, \mathit{Wait})$, $(60\,s, \mathit{Error})$ is expressed by the following SRL formula:

$$\varphi = t \; \underline{in}_{\exists} \; \exists(t = 6 \; \wedge (LOC[M] = \mathit{Wait}) \\ \wedge \exists(t = 60 \; \wedge (LOC[M] = \mathit{Error}) \wedge \mathtt{tt})),$$

and is matched, e.g., by the following (simplified) run:

$$\begin{aligned} & \langle \texttt{Off}, \{(\texttt{t}, 0)\}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ \rightarrow\ & \langle \texttt{Wait}, \{(\texttt{t}, 10)\}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ \rightarrow\ & \langle \texttt{Make\_Coffee}, \{(\texttt{t}, 15)\}, \{(\texttt{db}, 70), (\texttt{temp}, 35)\}\rangle \\ \rightarrow\ & \cdots \\ \rightarrow\ & \langle \texttt{Make\_Coffee}, \{(\texttt{t}, 59)\}, \{(\texttt{db}, 70), (\texttt{temp}, 100)\}\rangle \\ \rightarrow\ & \langle \texttt{Error}, \{(\texttt{t}, 60)\}, \{(\texttt{db}, 70), (\texttt{temp}, 100)\}\rangle \end{aligned}$$

### 4.7 Run transformation

When checking a model, we may not only be interested in whether an observation is matched by that model, but also in the actual model run that matches the observation. However, when checking reach() on the observation matcher system $M_m$, a model checker will return a run $r \in R(M_m)$, i.e., a run in the state space of $TS(M_m)$. To transfer such a run into the state space of the original timed automaton $M$, we need to remove the variables and intermediate states introduced by the simplifications and extensions performed in Sects. 4.5 and 4.6. We assume states $s$ of the following form for a parallel composition of model processes $p_1, p_2, \ldots, p_n$:

$$s := (s_{p1}{=}\langle l_{p1}, u_{p1}, v_{p1}\rangle \| \ldots \\ \ldots \| s_{pn}{=}\langle l_{pn}, u_{pn}, v_{pn}\rangle), \tag{26}$$

where $l_{pi}$ is the location state of $p_i$, $u_{pi}$ is the clock valuation of $p_i$, and $v_{pi}$ is the variable valuation in $p_i$. Furthermore, we assume runs of the following form:

$$r = s_1 \longrightarrow s_2 \longrightarrow \ldots \longrightarrow s_n. \tag{27}$$

Concretely, we can split the required transformation into the following 2 sub-transformations:

- $tf_s(r)$ removes the artificial states introduced in the original model (i.e., states where any intermediate location $l_{int}$ is active for committed state synchronization) and in the test automaton (i.e., states where $I_m, h, h_c$, or $m_T$ are active).
- $tf_p(r)$ removes the location, clock, and variable state of a given process from the parallel composition state.

These transformations build on the set of atomic transformations defined in the following. Given a run $r = \ldots s_a \longrightarrow s \longrightarrow s_b \ldots$, we define:

$$\mathit{rem\_state}(s, r) := \ldots s_a \longrightarrow s_b \ldots.$$

Given a state $s$ as defined in Eq. (26), we define:

$$\begin{aligned} & \mathit{rem\_proc}_{state}(p_i, s) := \\ & (s_{p1}{=}\langle l_{p1}, u_{p1}, v_{p1}\rangle \| \ldots \| \\ & s_{pi-1}{=}\langle l_{pi-1}, u_{pi-1}, v_{pi-1}\rangle \| \\ & s_{pi+1}{=}\langle l_{pi+1}, u_{pi+1}, v_{pi+1}\rangle \| \ldots \| \\ & s_{pn}{=}\langle l_{pn}, u_{pn}, v_{pn}\rangle), \end{aligned}$$

and for a run $r$ as defined in Eq. (27), we define:

$$\mathit{rem\_proc}(p, r) := \quad \mathit{rem\_proc}_{state}(p, s_1) \longrightarrow \ldots \\ \longrightarrow \mathit{rem\_proc}_{state}(p, s_n).$$

We can now define our 2 sub-transformations as follows:

$$\begin{aligned} & tf_s(r) := (\bigcirc_{s \in S_{rem}} \mathit{rem\_state}(s))(r), \\ & \text{with } S_{rem} = \{s | s = (l, u, v) \in r : l \in L_{rem}\} \\ & \qquad L_{rem} = \{l_{int}, I_m, h, h_c, m_T\} \\ & tf_p(r) := \mathit{rem\_proc}(\mathtt{T}, r) \end{aligned}$$

where $\bigcirc_{s \in S} f(s) = f(s_1) \circ \ldots \circ f(s_n)$ is the composition of functions $f(s_1), \ldots, f(s_n)$, and $\mathtt{T}$ is the test automaton. The complete transformation is then defined as:

$$tf = tf_s \circ tf_p. \tag{28}$$

To prove correctness of the model with regard to *tf*, we need to show that the following lemma holds:

**Lemma 10** (Preservation of runs) *Let $M$ be a timed automaton, $T$ be a test automaton, $M_m = M \| T$ be the composed matcher system, and $R(M_m)$ and $R(M)$ denote the sets of all runs of the transition systems $TS(M_m)$ and $TS(M)$. Then, $\forall r_m \in R(M_m) : \exists! r_o \in R(M) : tf(r_m) = r_o$, i.e., for every run $r_m$ in $R(M_m)$ exists exactly one run in $R(M)$ which $r_m$ is transformed to by tf.*

***Proof*** To show that Lemma 10 holds, we have to show that all transitions which were enabled in $TS(M_m)$, are also enabled in $TS(M)$ after their transformation. Each run $r_m$ in $M_m$ that ends in $(T, m_T)$, i.e., each run accepted by the test automaton, has the following general form:

$$r_m = r_{m,init} + \{r_{m,mc}, r_{m,sc}\}^* + r_{m,end},$$

where $r_{m,init}$ is the sub-run from the initial location $(T, I_m)$ to the first reaching of $(T, m_i)$, each $r_{m,mc}$ is a sub-run through the *matcher cycle* (i.e., the cycle $m_i \rightarrow h \rightarrow m_i$), each $r_{m,sc}$ is a sub-run through the *stepper cycle* (i.e., the cycle $m_i \rightarrow I_m \rightarrow m_i$), and $r_{m,end}$ is the final sub-run from $m_i$ over $h$ to $m_T$. We will show that each run $r_m$ is transformed into a run $tf(r_m) = r_o$ of the following form:

$$r_o = r_{o,init} + \{r_{o,mc}, r_{o,sc}\}^* + r_{o,end},$$

and that each $r_o$ is a run of $M$.

Exemplarily, the initial sub-run $r_{m,init}$ in the matcher system $M_m$ has the following form:

$$\begin{aligned} r_{m,init} = {} & (\langle \texttt{l\_init}, u_{o,init}, v_{o,init}\rangle \\ & \| \langle \texttt{I\_m}, u_{m,init}, \{(\texttt{i}, 0), (\texttt{\_stepped}, true)\}\rangle) \\ \xrightarrow{\tau} {} & (\langle \texttt{l\_init}, u_{o,init}, v_{o,init}\rangle \\ & \| \langle \texttt{m\_i}, u_{m,init}, \{(\texttt{i}, 0), (\texttt{\_stepped}, false)\}\rangle), \end{aligned}$$

and is transformed to the sub-run $r_{o,init}$ as follows:

$$r_{o,init} = tf(r_{m,init}) = \langle \texttt{l\_init}, u_{o,init}, v_{o,init}\rangle,$$

which is simply the initial state of $M$. For the complete proof including all run parts $r_{o,init}$, $r_{m,mc}$, $r_{m,sc}$, and $r_{m,end}$, see Sect. B.6. □

While Lemma 10 shows that runs in the matcher system $M_m$ can be transformed into runs of the original model $M$, it makes no statement on the preservation of the observation matching property. Thus, we also need to prove that the following lemma holds:

**Lemma 11** (Transfer observation matching in transformation) *Let $M$ be a timed automaton, $T$ be a test automaton, $M_m = M \| T$ be the composed matcher system, $R(M_m)$ denote the set of all runs of $M_m$, and $\varphi(OBS)$ be an SRL property derived from a given observation $OBS$. Then, $\forall r \in R(M_m) : (r \models \varphi(OBS)) \implies (tf(r) \models \varphi(OBS))$.*

***Proof*** $r \models \varphi(OBS)$ implies that for each $obs \in OBS$ there exists a state $s$ in $r$ that matches $obs$ (cf. the Lemma 5), and these matchings occur in order. The transformation *tf* keeps all states with $m_i$ and $m_{i,c}$ as active locations, i.e., those states in which the constraints of the observation points are enforced, and does not alter the state portions of the original model $M$ in states on $r$. Therefore, the states sections that matched the observations in $OBS$ remain intact, from which we conclude that $tf(r) \models \varphi(OBS)$, and thus, Lemma 11 holds. □

***Example 12*** (Run transformation) For the vending machine (Fig. 1), we may get the following run (with the extended variable vector $(db, temp, i, \_stepped)$ for each state) when we match the observation sequence $(0\,s, \texttt{Off}), (5\,s, \texttt{Wait})$:

$$\begin{aligned} r = {} & (\langle \texttt{Off}, u_{o,0}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{I\_m}, u_{m,0}, \{(\texttt{i}, 0), (\texttt{\_stepped}, true)\}\rangle) \\ \rightarrow {} & (\langle \texttt{Off}, u_{o,0}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{m\_i}, u_{m,0}, \{(\texttt{i}, 0), (\texttt{\_stepped}, false)\}\rangle) \\ \rightarrow {} & (\langle \texttt{Off}, u_{o,0}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{h}, u_{m,0}, \{(\texttt{i}, 1), (\texttt{\_stepped}, false)\}\rangle) \\ \rightarrow {} & (\langle \texttt{Off}, u_{o,0}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{m\_i}, u_{m,0}, \{(\texttt{i}, 1), (\texttt{\_stepped}, false)\}\rangle) \\ \rightarrow {} & (\langle \texttt{l\_int}, u_{o,1}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, 1), (\texttt{\_stepped}, true)\}\rangle) \\ \rightarrow {} & (\langle \texttt{l\_int}, u_{o,1}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{I\_m}, u_{m,1}, \{(\texttt{i}, 1), (\texttt{\_stepped}, true)\}\rangle) \\ \rightarrow {} & (\langle \texttt{Wait}, u_{o,1}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, 1), (\texttt{\_stepped}, false)\}\rangle) \\ \rightarrow {} & (\langle \texttt{Wait}, u_{o,1}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{h}, u_{m,1}, \{(\texttt{i}, 2), (\texttt{\_stepped}, false)\}\rangle) \\ \rightarrow {} & (\langle \texttt{Wait}, u_{o,1}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \| \langle \texttt{m\_T}, u_{m,1}, \{(\texttt{i}, 2), (\texttt{\_stepped}, false)\}\rangle) \end{aligned}$$

Applying the transformation *tf* to the run $r$ removes the artificial variables and states as well as the test automaton component, and we get:

$$\begin{aligned} tf(r) = {} & \langle (\texttt{Off}), u_{o,0}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle \\ & \rightarrow \langle (\texttt{Wait}), u_{o,1}, \{(\texttt{db}, 0), (\texttt{temp}, 20)\}\rangle, \end{aligned}$$

which is a run in the original vending machine model (Lemma 10). As furthermore the two constraints $(t = 0)(u_{o,0})$ and $(t = 5)(u_{o,1})$ hold, the transformed run is also a matching run (Lemma 11).

### 4.8 Results and discussion

The following theorem summarizes the theoretical results that we obtained in this section for observation matching via matcher systems:

**Theorem 1** (Correct observation matching with matcher systems) *Let $M$ be a timed automaton, $OBS$ be a (possibly partial, imprecise or delayed) observation, $T$ be the test automaton derived from $OBS$, $M_m = M \| T$ be the matcher system with acceptance node $m_T$, $TS(M)$ and $TS(M_m)$ be the transition systems, and $tf$ be the transformation from matcher system runs $r_m$ to original runs $r$. Then:*

$$\forall r_m \in R(M_m) : reach(r_m, m_T) \iff match(r = tf(r_m), OBS). \tag{29}$$

In other words, the reachability analysis of the acceptance node $m_T$ in the matcher system $M_m$ (cf. Sections 4.4, 4.5 and 4.6) correctly identifies runs $r_m$ which correspond to runs $r$ in $M$ that match $OBS$ under the matching criteria introduced in Sect. 4.1.

***Proof*** From Lemma 6 (which holds for the base formulae of Lemma 5 and Lemma 4 as well as the extended formulae of Lemma 8 and Lemma 9) we know that a test automaton $T$ that correctly models observation matching can be constructed for an observation $OBS$, and thus, the reachability of $m_T$ in $M_m = M \| T$ yields matching runs in the domain of $M_m$. From Lemma 10, we further know that corresponding runs in the domain $M$ can be obtained by transformation (Eq. (28)). Finally, from Lemma 11, we know that the observation matching property is preserved during the transformation. Thus, Theorem 1 holds. □

A few remarks should be made on the results of this section: First, the observability of intermediate states needs to be considered in advance of selecting a suitable test automaton. In the case of our example model (cf. Figure 1), it is a reasonable decision to allow the observation of committed locations, as they are used for the simple indication of error states. One may come to a different conclusion when variable assignments are involved on multiple edges of such a committed section. Then, one needs to decide which data is actually observable if a model yields different variable values in the same time instant; using only the final value of committed sections may be sensible then, omitting the necessity of committed location matching.

Second, one may consider to implement the observation sequence traversal as separate process in the model. Right now, the sequence is handled as array in the code declaration (cf. Figure 5d and the model transformation in Fig. 8b), and the steps through that array are performed in the same sub-model that also performs the actual matching (cf. Figure 5c). One may argue that these concerns should be separated. Such separation could be achieved by modeling the observation sequence in a different sub-model, which assigns the observation points sequentially to dedicated observation variables (e.g., $obs\_x$), against which the test automaton then compares the values of the actual variables (e.g., $x == obs\_x$).

Third, cases exist in which case-dependent optimizations of the matcher model are reasonable. As an example, in a model with large sections of urgent transitions (i.e., sections where no time passes), each intermediate state is considered in the matching step, which leads to excessive branching in the run space even though the concrete state used during matching does not affect the overall matching outcome. In such case, one might enforce steps in the matcher as soon as they are enabled, e.g., by using channel priorities on synchronizations for the matcher and stepper cycle, where the matcher is prioritized whenever possible.

## 5 Implementation

We provide a Python implementation of the observation matching approach for Uppaal models, which is released as open source software [32] under the MIT license, and which consists of the following modules:

- The Uppaal TA implementation, the Uppaal C code parser (based on the original Uppaal BNF grammar), and the Uppaal trace parser.
- The matcher system generator, which transforms the original model to the matcher system by integrating the matcher model and adding helper structures and observation data as needed.
- The run transformer, which reduces runs of the matcher model to runs of the original model.
- The observation generator, which determines valid runs of the given base model, and extracts observation data points from these runs for our experiments.
- The experiments performed for this article.

After installation, executing the command `run` in the command-line interface of the experiments package will start all experiments and store the resulting data. The experiments are available online [33], and the experiment setup and results are explained in Sect. 6. In contrast to Sect. 4, where we introduced matcher models for both observation sequences and observation traces, the implementation and

**Algorithm 1** The matcher system generation.

**Input** : Original model $M$, observation data $OBS$, options $opt$
**Output:** The composed matcher system $M_m$
1 $M_p \leftarrow preprocess\_model(M)$
2 **if** $opt$["$location\_matching$"] **then**
3 $\quad M_p \leftarrow enable\_location\_matching(M_p)$
4 **if** $opt$["$committed\_matching$"] **then**
5 $\quad M_p \leftarrow enable\_committed\_matching(M_p)$
6 $M_t \leftarrow$
$\quad select\_matcher\_template(opt["shifted\_matching"],$
$\quad\quad opt["committed\_matching"])$
7 $M_m \leftarrow (M_p \| M_t)$
8 $M_m \leftarrow insert\_observation(M_m, OBS)$
9 $M_m \leftarrow update\_observation\_check(M_m, opt["partial\_matching"])$
10 $M_m \leftarrow add\_query(M_m, \texttt{E<> Matcher.T})$
11 **return** $M_m$

**Algorithm 2** The observation generation.

**Input** : Original model $M$, options $opt$
**Output:** The observation data $OBS$
1 $M_p \leftarrow add\_step\_counter\_and\_helper\_clocks(M)$
2 $M_p \leftarrow add\_helper\_locs\_for\_intermediate\_clock\_states(M_p)$
3 $M_p \leftarrow add\_query(M_p, \texttt{E<> step == opt["step\_bound"]})$
4 $tr_{sym} \leftarrow obtain\_symbolic\_run(M_p)$
5 $tr_{sym,det} \leftarrow obtain\_deterministic\_symbolic\_run(tr_{sym})$
6 $OBS \leftarrow extract\_observation\_from\_run(tr_{sym,det})$
7 $OBS_m \leftarrow manipulate\_observation(OBS, opt)$
8 **return** $OBS_m$

experiments focus only on observation sequences, as their matching supports all extensions introduced in Sect. 4.6, and their experimental evaluation is more comparable between models due to the fact that states can be observed after each step, whereas some models in the experiment model suite contain only scattered or no observable actions at all.

### 5.1 Matcher system generator

The core routine of the *matcher system generator* is depicted as pseudocode in Algorithm 1. The matcher takes an original model and observation data as well as feature settings (e.g., allowed deviation, potential delay, inclusion of committed states) as input, and composes the matcher system accordingly. First, a model preprocessing step turns all processes into explicit templates, and moves all variables into the global declaration scope to make their data accessible from the matcher model (l.1). Then, the original model sections are extended for active (l.2–3) and committed (l.4–5) location tracking if specified. Depending on the delay and check for committed states, a different base matcher template is selected (l.6) and composed with the original model (l.7), and in case of committed matching, synchronization calls between the original model and the matcher model are established. Finally, the code declaration and matcher model are updated with the observation data (l.8), the variables checks are adapted accordingly (l.9), and the reachability query is added (l.10).

### 5.2 Run transformer

In Sect. 4, we introduced observation matching and proved its correctness based on a concrete interpretation of states and runs. Uppaal, however, uses an abstract interpretation, where the time domain is represented symbolically in the form of *difference bound matrices (DBMs)* [34] instead of concrete valuations, which makes the state and run space enumerable under certain conditions. One can view the symbolic interpretation of states as grouping of a (potentially infinite) number of consecutive concrete states of a run which share the same location and variable valuation, i.e., states reached by consecutive delay transitions. Likewise, a symbolic run can be viewed as a (potentially infinite) number of concrete runs over the same sequence of triggered edges.

Given that all concrete runs included in such symbolic run are actual runs of the model, the results of Sect. 4 are applicable in the symbolic case as the following two properties hold:

1. Each concrete model run satisfying the $SRL$ formula $\varphi_{SRL}$ is included in a symbolic model run satisfying the same formula, i.e., $\forall r_{con} \in TR_{con}(M) : r_{con} \models \varphi_{SRL} \implies (\exists r_{sym} \in TR_{sym}(M) : r_{sym} \models \varphi_{SRL} \wedge r_{con} \in r_{sym})$.
2. For each symbolic model run satisfying the $SRL$ formula $\varphi_{SRL}$, all concrete model runs included in that symbolic run satisfy the same formula, i.e., $\forall r_{sym} \in TR_{sym}(M) : r_{sym} \models \varphi_{SRL} \implies (\forall r_{con} \in r_{sym} : r_{con} \models \varphi_{SRL} \wedge r_{con} \in TR_{con}(M))$.

Property (1) holds as for each concrete model run, there exists a symbolic run in the model that contains only that particular run; the zones described by the symbolic clock states of the run are single points then. Property (2) holds as the clock zones of a symbolic run satisfying $\varphi_{SRL}$ span sub-intervals of the clock intervals described by $\varphi_{SRL}$, and thus, the time valuations of all concrete runs included in the symbolic run necessarily lie in these intervals as well, satisfying the formula.

The symbolic run obtained from the matcher model needs to be transformed into the corresponding symbolic run of the original model. The *run transformer* applies the transformations as specified in Sect. 4.7, and furthermore implements the check for symbolic sub-run relations, i.e., checking if the set of concrete runs of a symbolic run is a subset of the concrete runs of another symbolic run.

### 5.3 Observation generator

The *observation generation* routine is outlined in Algorithm 2. From an original model and settings regarding the desired amount of transitions to take and data manipulations to apply, the routine simulates a random run through the model and extracts observation states from that run. The model is first preprocessed by adding a *global clock* ($t_g$) which is never reset and thus represents global time, a *reset clock* ($t_r$) which is reset on every original edge, and a *step counter* which is incremented on every original transition, i.e., on every edge that is either non-synchronized or involved as caller (via `!`) in a synchronization ($l.1$). Furthermore, every such non-synchronized or "calling" edge is split into two edges over committed helper locations ($l.2$), where the first edge applies the synchronization and guards of the original edge, and the second edge applies its clock resets and variable updates. The reset clock and the edge splitting add additionally required information on active times of locations and intermediate delayed states, respectively, to the simulated runs for the extraction of observations from $r_{sym}$ later on. The query `E<> step == opt["step_bound"]` is added (l.3), which checks for a symbolic run on which the step counter reaches the desired bound (l.4). From that run, concrete observation states are extracted (l.5–6), which can also be manipulated optionally based on the observation settings (l.7), i.e.:

- single variable values of individual data points may be removed for partial observations,
- individual time and variable values may be shifted for value deviation,
- all time values may be shifted for initial delay, and
- data points with active committed locations may be removed if committed states should be unobservable.

The extraction of concrete observation states from the symbolic run $r_{sym}$ is split into two steps: The `obtain_deterministic_symbolic_run` step enforces concrete leaving times on the states of $r_{sym}$, which makes the run deterministic regarding the time spent in each location state. That way, we obtain new symbolic states from which we can easily extract observation states. The `extract_observation_from_run` step performs that extraction by repeatedly selecting an arbitrary symbolic states from the run, then choosing an arbitrary time $t$ from the interval of valid global clock values from its symbolic clock state, and combining that clock value $t$ with the corresponding location and variable state.

## 6 Empirical evaluation

In this section, we perform an empirical evaluation of the observation matching approach described in Sect. 4 and its concrete implementation described in Sect. 5. For that purpose, we perform a set of experiment runs, each performing matching of a given observation on a model based on a particular matcher system. An experiment run is fully specified by the following 5 parameters:

- The *type of matcher model*, i.e., raw matcher model (`R`, Sect. 4.4), simplified base matcher model (`B`, Sect. 4.5), base matcher extended for partial (`BP`), deviating (`BD`), location-based (`BL`), time-shifted (`BS`), or committed (`BC`) observation matching (Sect. 4.6), or all extensions combined (`All`).
- The *type of observation*, i.e., partial (`P`), deviating (`D`), location-based (`L`), time-shifted (`S`), or committed (`C`) observations, basic observations (`B`) with none of the aforementioned attributes, or observations combining all attributes (`All`).
- The *number of observation points*, i.e., `few` or `many` observations.
- The *observation time range*, i.e., `short`- or `long`-term observations.
- The *containment classification*, i.e., if the observation is actually contained in the model (`+`) or not (`-`).

For example, the type `(BS, B, few, long, +)` specifies that the base matcher model extended for time-shifted matching (`BS`) is used to match a basic observation (`B`, i.e., a full and exact, non-shifted observation of variable data in non-committed states only) consisting of sporadic observation points (`few`) over a long time range (`long`), and the observation is contained in the model (`+`).

Using these configurations, it is expectable that rather basic matching scenarios (e.g., `(B, B, few, short, +)`) will require way shorter run times than more complex scenarios (e.g., `(All, All, many, long, +)`), where the most complex matcher model instance is applied to extensive observations sequences with a variety of imprecisions. For a detailed experimental evaluation of the 5 parameters, the following research questions arise in terms of *applicability* and *performance* of observation matching:

(AQ1) Is the implementation described in Sect. 5 applicable for correct matching of observations as described in Sect. 4?
(PQ1) Do the simplifications applied in Sect. 4.5 result in a model with improved matching performance?
(PQ2) Can the extended matcher models introduced in Sect. 4.6 be used in place of the more basic matcher

models to match basic observations without overhead?

(PQ3) Does the observation type impact the matching performance of a given matcher model?

(PQ4) How does the matching run time relate to the number of observations and transitions?

In particular, (AQ1) correctly classifies observations as contained or non-contained, (PQ1) assesses if the simplifications—especially those affecting the state space—also benefit matching besides readability, (PQ2) investigates if the most extended matcher model version alone may be a suitable choice for all matching scenarios, and (PQ3) and (PQ4) investigate if certain observations may result in exceedingly long matching times.

## 6.1 Baseline

To the best of our knowledge, there does not exist a viable baseline that our approach of matching imprecise observations on TAs could be compared against in a meaningful way. One may think about using standard TCTL formulae or the concrete simulator implemented in Uppaal; however, series of observation points cannot be expressed with the TCTL formulae supported by Uppaal (so that one would be restricted to a single observation point), and the concrete simulator does not allow for non-exact matches as it only simulates transitions that exist in the model. Similarly, using related tools such as Uppaal Tron would also per se only allow for exact matching (with the difference that the observations would then be automatically generated by a modeled "system under test"), and any sort of non-exact matching would again require model transformations such as those described in this article. Therefore, as none of these approaches allow for comparable results, we interpret the results of our approach "absolutely" in terms of applicability and performance without comparison to a baseline.

## 6.2 Setup

We execute all experiments on an Ubuntu 20.04 LTS system with AMD Ryzen 7 2700X eight-core CPU and 16 GB RAM. Our experiment model suite consists of 10 different models: The model `main-example-model` is the model from the introduction, and the 7 models `2doors`, `bridge`, `fischer`, `fischer-symmetry`, `interrupt`, `train-gate`, and `train-gate-orig` are the demo model suite of standard Uppaal excluding models based on live sequence charts. The models `csmacd2` [35] and `tdma` [36] were developed in case studies. We adapted the models as follows: In `fischer` and `fischer-symmetry`, we reduced the number of processes from 6 and 10, respectively, to 3, as the explicit splitting of processes into templates in the preprocessing step hinders optimization by symmetry reduction; a problem that requires further inspection in future, e.g., by omitting the explicit splitting and accessing the variables that we moved from local to global space by indexing instead. Furthermore, we added helper variables to `csmacd2` to allow for observable variable states.

The model suite allows us to apply all state-related concepts introduced in this article, i.e., they expose global clocks, several data variables which we can interpret as either observable or non-observable (among which some hold integer values for which we can express imprecisions), and both named and committed locations. Note that we exclude observation traces in the experiments on purpose, as they expose only a subset of attributes (i.e., time deviations and shifts) that are also exposed by observation sequences, and thus, would not add any additional insights for the evaluation and conclusion.

### 6.2.1 Numbers of experiments

In the classification experiment, we apply the matching routine to 1000 positive and negative observations for each model, which we consider a suitable number of runs to get an intuition that the classification works as intended. In the experiments for different matcher model versions and observation types, we perform 100 runs per experiment configuration (with $360 + 70$ different configurations in total), which provides enough data to interpret individual numbers for the matcher run time. Finally, in the experiments with variable observation sizes and time ranges, we execute a smaller number of 20 runs each for a total of $40 + 40$ experiment configurations (note here that the runs take increasing amounts of time with growing observation sizes and transition counts), providing us with sufficient data to give an insight into the overall trends of matcher run times (Table 2).

### 6.2.2 Observation generation

For the generation of positive observations, we use the original model both for generation of observations (which are manipulated afterward toward particular observation types) and matching of said observations to make sure that the observations are indeed contained in the model. The full workflow of a single experiment run for matching of positive observations consists of the following 6 steps:

1. Derive a symbolic run and concrete observation data from the original model.
2. Compose the matcher model based on observation data and settings.
3. Determine a *matching symbolic run* with the matcher model.

**Table 2** The numbers of correct and incorrect matches among 1000 positive and 1000 negative observation sequences for each model

| Model name | Positives | | Negatives | |
|---|---|---|---|---|
| | True | False | True | False |
| `main-example-model` | 1000 | 0 | 1000 | 0 |
| `2doors` | 1000 | 0 | 1000 | 0 |
| `bridge` | 1000 | 0 | 1000 | 0 |
| `fischer` | 1000 | 0 | 1000 | 0 |
| `fischer-symmetry` | 1000 | 0 | 1000 | 0 |
| `interrupt` | 1000 | 0 | 1000 | 0 |
| `train-gate` | 1000 | 0 | 1000 | 0 |
| `train-gate-orig` | 1000 | 0 | 1000 | 0 |
| `csmacd2` | 1000 | 0 | 1000 | 0 |
| `tdma` | 1000 | 0 | 1000 | 0 |

4. Derive the *transformed symbolic run* in the original model domain from the matching symbolic run.
5. Determine the *actual symbolic run* by simulation of the original model based on the actions / transitions of the transformed symbolic run.
6. Check if the transformed symbolic run of Step 4 is contained in the actual symbolic run of Step 5.

Generating negative observation sequences, in contrast, constitutes an infeasible task, as negatives cannot be derived from the model itself in the general case. Thus, we restrict the automated generation of negative observations for our experiment model suite to those cases where containment is trivially decided, e.g., with observation values out of defined bounds, or clock values smaller than the values of previous observation points.

For the scope of these experiments, we use the transition count of the simulated model (from which we derive the observations) as fourth parameter instead of the observation time range; a reasonable decision considering that the states are symbolic in Uppaal, and thus, a single state can already span potentially infinite time ranges. We control the number of transitions via a counter variable as described in Sect. 5.3, and derive zero, one, or more concrete observations from each of the reached symbolic states, up to the number of observation points set for the experiment run.

### 6.3 Results

Table 2 shows the numbers of correctly and incorrectly classified positive and negative observations in a total of 1000 positives and 1000 negatives for each model. Table 3 shows the average matching times for basic observations of different lengths and transition counts for all matcher model versions applied to all models of the experiment model suite, and Table 4 shows the matching times for different observation types using the most complex matcher model; in both figures, the values referred to in the evaluation text are highlighted in bold. Finally, the graphs in Fig. 11 show the matching durations in relation to growing observation sizes and transition counts, respectively (Table 3).

### 6.4 Evaluation

Table 2 provides an answer to the research question (AQ1); in all cases, positives and negatives were correctly classified. The experiment results allow us to conclude that the implementation correctly realizes the theoretical results, and they empirically confirm that the theory itself is correct, as otherwise, misclassifications would occur here (Table 4).

The verification durations in Table 3 answers the research questions (PQ1) and (PQ2). In particular, we see that the raw matcher (`R`) requires strictly more time for matching than the simplified base matcher model (`B`), which is most noticeable for the `train-gate` and `tdma` models in the case of `many,long` observations, where the raw matcher takes averages of 22.723 s and $>$ 30 s, while the base matcher requires 0.533 s and 0.147 s, respectively. The main reason for that difference is that the raw matcher handles upper clock constraints as guards instead of invariants, which leads to explicit checking of run suffixes which are already known to not satisfy the constraints. Furthermore, we see that the most extended matcher model takes at most around 3 times as much time as the base matcher model (e.g., 1.768 s compared to 0.533 s for `many,long` observations in `train-gate`), and observe that especially the extensions for committed matching (`BC`), and time-shifted matching in the context of committed matching (`BSC`, cf. Section 4.6.3) require the most time for matching basic observations. The reason for the overhead of `BC` and `BSC` lies in the additional transitions per stepper cycle introduced by the synchronization of model and test automaton. The performance impact of the remaining extensions (cmp. `BSC` to `All`) is negligible in contrast, which is expected as these extensions affect neither the state space nor the complexity of the state matcher checks. Overall, we conclude that the simplifications indeed improve matching performance significantly ((PQ1)), and that it matters which matcher model is chosen for the matching of basic observations, where especially the feature of committed location matching should only be enabled when observations require it ((PQ2)).

The research question (PQ3) is answered by the data in Table 4. In general, the routines for additional features, e.g., the tracking of active or committed locations, are executed if they are implemented in the model, independent of whether the observation actually exhibits the corresponding traits. The matching times highly depend on the individual models:

**Table 3** The matcher run time for basic observations of different sequence lengths and transition counts averaged over 100 runs with all matcher model versions

| Model name | Observation | Matching run Time (averaged over 100 runs) [s] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | R | B | BP | BD | BL | BS | BC | BSC | All |
| main-example-model | few.short | 0.020 | 0.013 | 0.014 | 0.014 | 0.015 | 0.014 | 0.018 | 0.018 | 0.019 |
| | many.short | 0.038 | 0.015 | 0.015 | 0.014 | 0.015 | 0.015 | 0.019 | 0.019 | 0.020 |
| | few.long | 0.075 | 0.017 | 0.017 | 0.016 | 0.017 | 0.017 | 0.024 | 0.024 | 0.025 |
| | many.long | 0.206 | 0.018 | 0.018 | 0.018 | 0.018 | 0.018 | 0.024 | 0.025 | 0.025 |
| 2doors | few.short | 0.022 | 0.014 | 0.014 | 0.014 | 0.016 | 0.015 | 0.019 | 0.019 | 0.021 |
| | many.short | 0.036 | 0.016 | 0.016 | 0.015 | 0.016 | 0.016 | 0.020 | 0.021 | 0.022 |
| | few.long | 0.030 | 0.017 | 0.017 | 0.016 | 0.018 | 0.017 | 0.023 | 0.024 | 0.025 |
| | many.long | 0.045 | 0.019 | 0.019 | 0.019 | 0.020 | 0.019 | 0.026 | 0.027 | 0.027 |
| bridge | few.short | 0.054 | 0.017 | 0.017 | 0.017 | 0.018 | 0.017 | 0.021 | 0.022 | 0.022 |
| | many.short | 0.126 | 0.019 | 0.019 | 0.019 | 0.019 | 0.019 | 0.024 | 0.025 | 0.025 |
| | few.long | 0.069 | 0.023 | 0.023 | 0.023 | 0.024 | 0.024 | 0.032 | 0.032 | 0.033 |
| | many.long | 0.150 | 0.025 | 0.025 | 0.025 | 0.026 | 0.026 | 0.030 | 0.029 | 0.031 |
| fischer | few.short | 0.025 | 0.013 | 0.013 | 0.013 | 0.013 | 0.013 | 0.016 | 0.018 | 0.018 |
| | many.short | 0.047 | 0.015 | 0.015 | 0.015 | 0.016 | 0.016 | 0.021 | 0.022 | 0.023 |
| | few.long | 0.029 | 0.016 | 0.016 | 0.015 | 0.016 | 0.015 | 0.019 | 0.019 | 0.021 |
| | many.long | 0.058 | 0.021 | 0.021 | 0.021 | 0.021 | 0.021 | 0.045 | 0.045 | 0.046 |
| fischer-symmetry | few.short | 0.030 | 0.014 | 0.015 | 0.015 | 0.015 | 0.015 | 0.019 | 0.019 | 0.019 |
| | many.short | 0.054 | 0.016 | 0.015 | 0.015 | 0.017 | 0.016 | 0.022 | 0.022 | 0.023 |
| | few.long | 0.035 | 0.017 | 0.018 | 0.017 | 0.018 | 0.017 | 0.027 | 0.028 | 0.029 |
| | many.long | 0.059 | 0.020 | 0.020 | 0.020 | 0.022 | 0.020 | 0.038 | 0.040 | 0.041 |
| interrupt | few.short | 0.019 | 0.012 | 0.012 | 0.012 | 0.013 | 0.013 | 0.016 | 0.016 | 0.016 |
| | many.short | 0.029 | 0.015 | 0.015 | 0.015 | 0.015 | 0.015 | 0.018 | 0.020 | 0.020 |
| | few.long | 0.018 | 0.012 | 0.013 | 0.012 | 0.013 | 0.013 | 0.016 | 0.016 | 0.017 |
| | many.long | 0.029 | 0.015 | 0.015 | 0.014 | 0.015 | 0.015 | 0.019 | 0.021 | 0.021 |
| train-gate | few.short | 4.893 | 0.070 | 0.069 | 0.070 | 0.072 | 0.076 | 0.137 | 0.179 | 0.177 |
| | many.short | 8.931 | 0.109 | 0.106 | 0.107 | 0.111 | 0.118 | 0.249 | 0.304 | 0.300 |
| | few.long | 8.892 | 0.539 | 0.541 | 0.541 | 0.547 | 0.549 | 1.389 | 1.437 | 1.439 |
| | many.long | **22.723** | **0.533** | 0.536 | 0.532 | 0.540 | 0.545 | 1.684 | 1.728 | **1.768** |
| train-gate-orig | few.short | 0.101 | 0.018 | 0.018 | 0.018 | 0.019 | 0.018 | 0.026 | 0.029 | 0.030 |
| | many.short | 0.133 | 0.018 | 0.018 | 0.018 | 0.020 | 0.020 | 0.026 | 0.029 | 0.031 |
| | few.long | 0.203 | 0.027 | 0.028 | 0.027 | 0.028 | 0.028 | 0.050 | 0.053 | 0.055 |
| | many.long | 0.234 | 0.021 | 0.022 | 0.021 | 0.023 | 0.021 | 0.033 | 0.034 | 0.036 |
| csmacd2 | few.short | 0.019 | 0.014 | 0.015 | 0.015 | 0.016 | 0.015 | 0.019 | 0.020 | 0.021 |
| | many.short | 0.030 | 0.016 | 0.017 | 0.017 | 0.018 | 0.017 | 0.022 | 0.022 | 0.024 |
| | few.long | 0.018 | 0.015 | 0.015 | 0.014 | 0.016 | 0.014 | 0.019 | 0.019 | 0.021 |
| | many.long | 0.024 | 0.015 | 0.016 | 0.015 | 0.017 | 0.016 | 0.021 | 0.022 | 0.023 |
| tdma | few.short | 0.103 | 0.038 | 0.039 | 0.039 | 0.046 | 0.039 | 0.069 | 0.069 | 0.081 |
| | many.short | 0.486 | 0.046 | 0.045 | 0.047 | 0.051 | 0.046 | 0.063 | 0.063 | 0.074 |
| | few.long | 0.538 | 0.054 | 0.054 | 0.054 | 0.060 | 0.054 | 0.114 | 0.115 | 0.126 |
| | many.long | **>30** | **0.147** | 0.150 | 0.148 | 0.152 | 0.149 | 0.331 | 0.333 | 0.349 |

The values referred to in the evaluation text are highlighted in bold

**Table 4** The matcher run time for different observation types with fixed sequence length and transition count averaged over 100 runs using the most complex matcher model version

| Model name | Matching run time (averaged over 100 runs) [s] | | | | | | |
|---|---|---|---|---|---|---|---|
| | B | P | D | L | S | C | All |
| `main-example-model` | 0.035 | 0.037 | 0.035 | 0.029 | 0.033 | 0.035 | 0.032 |
| `2doors` | 0.026 | 0.026 | 0.031 | 0.030 | 0.026 | 0.026 | 0.030 |
| `bridge` | 0.028 | 0.028 | 0.030 | 0.043 | 0.028 | 0.027 | 0.043 |
| `fischer` | 0.032 | 0.033 | 0.045 | 0.053 | 0.032 | 0.032 | 0.053 |
| `fischer-symmetry` | 0.034 | 0.034 | 0.051 | 0.061 | 0.034 | 0.034 | 0.060 |
| `interrupt` | 0.021 | 0.021 | 0.022 | 0.022 | 0.021 | 0.021 | 0.023 |
| `train-gate` | **0.329** | 0.317 | **6.283** | **1.373** | 0.352 | 0.348 | **1.044** |
| `train-gate-orig` | 0.029 | 0.029 | 0.054 | 0.065 | 0.029 | 0.039 | 0.070 |
| `csmacd2` | 0.027 | 0.027 | 0.028 | 0.029 | 0.027 | 0.027 | 0.029 |
| `tdma` | **1.107** | 1.098 | 1.092 | 0.092 | **1.697** | **0.838** | 0.092 |

The values referred to in the evaluation text are highlighted in bold

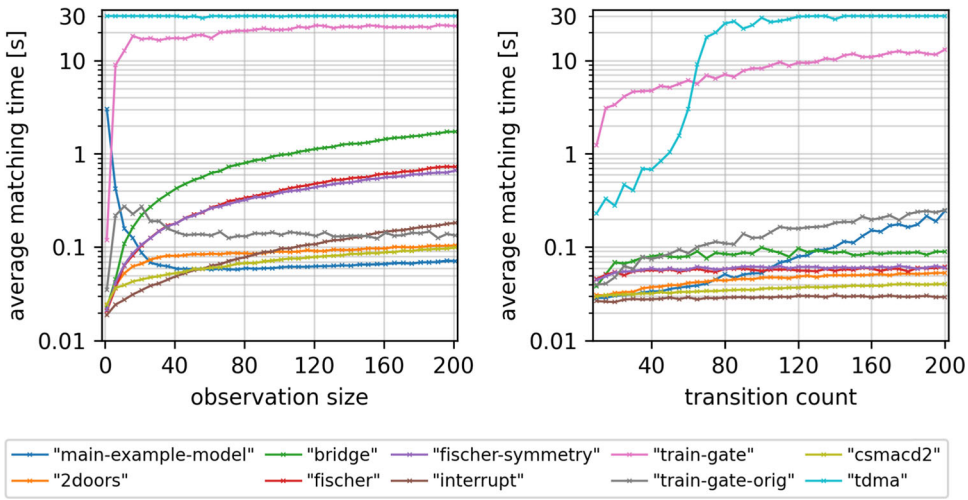



**Fig. 11** The matcher run time for different observation sequence lengths (left) and transition counts (right) averaged over 20 runs using the most complex matcher model version

Deviating data often leads to worse performance, most noticeable in `train-gate` (6.283 s compared to 0.329 for $B$), where value deviation allows for many potential runs in the beginning, and only toward the end of the observation sequence restricts to only a few possible runs, so that the remaining runs are only discarded at a late stage. For other models (e.g., `main-example-model`, `interrupt`, `csmacd2`, and `tdma`), the performance stays comparable to other observation types.

The observability of locations results in better performance for `main-example-model` and, most noticeable, for `tdma` (0.092 s compared to 1.107 s for $B$), where additional location data leads to early discarding of runs whose initial sections would otherwise match the observed time and variable data at first before starting to deviate from the observation. For the remaining models, the performance becomes worse, especially for `train-gate` (1.373 s compared to 0.329 for $B$), where we only observe the length of the Gate queue of arriving trains in our experiment, which is equal for different arrival orders. If locations are observable here at scattered steps, they force longer evaluation of runs before discarding, as they partly dictate which train order is eventually accepted, without providing enough information on that early on; without location data, the observable queue length may be matched by several different train orders.

Time-shifted and committed observations do not lead to noticeable effects on the performance for most experiment models, with one exception: In the `tdma` model, time-shifted data leads to increased matching times (1.697 s compared to 1.107 s in $B$), as the model contains a long initial section with potentially no time value change, resulting in later discarding of runs after each potential delay. Furthermore, committed observations here lead to better performance (0.838 s compared to 1.107 s in $B$), as the model has a noticeable amount of committed locations after branching points, whose observability allows for an earlier acceptance or discarding of runs.

Combining all observation types (`All`) averages the impacts of the individual types in most cases, and except for `train-gate` (with a matching run time of 1.044 s on average), all resulting averages lie below 1 s. In summary, two

effects interfere in terms of imprecisions (i.e., deviations and potential time-shifts): While imprecision may lead to earlier acceptance of imprecisely matched paths, which decreases the run time, it may also lead to belated discarding of eventually non-matching paths, which in turn increases the run time. The experiments results imply that the latter effect usually predominates, leading to overall higher run times. Thus, we conclude that the observation types impact the matching performance ((PQ3)) negatively especially when they impose additional imprecision, and may have a positive impact when more precise data (e.g., concrete locations) is observed.

Finally, research question (PQ4) is answered by the graphs in Fig. 11. Overall, we see that the matching duration tends to rise with an increasing number of observation points (left figure); however, especially in the section of fewer observation counts, the `main-example-model`, `train-gate`, and `train-gate-orig` models behave different from the remaining models (also note for `tdma` that the duration of each run was capped at 30 s): In the case of `main-example-model`, the average matching time first decreases in the range between 1 and 50 observations, before it slowly increases again with more than 50 observations. For `train-gate`, increasing the observation count from 1 to 10 leads to a strongly increased matching time (from ∼0.1 to ∼15 s), and only with more observations behaves like the remaining models. The duration times of `train-gate-orig` finally show traits of both aforementioned models. A reason for the observed characteristics could be that fewer observations enforce fewer constraints on a matching run, so that a match is found earlier; more observations in contrast enforce additional constraints, and require additional matching steps. Additional observations, however, may also allow to discard non-matching runs earlier, which may be the most influential aspect for the smaller observation counts of `main-example-model`. For the transition counts (right figure), we observe similarly that the matching time rises with increasing numbers of transition, which is expected due to growing run lengths, and no considerable outliers occurred. We conclude that the matching run times, as expected, are mostly negatively affected by increasing observation sequence lengths and transition counts ((PQ4)) in the long run, even though the effects in the short run may be positive when the benefit of early discarding of runs predominates.

### 6.5 Summary

In summary, we could show that all matcher model implementations work correctly. However, in terms of performance, we see that even though the most complex matcher model is applicable to all covered types of observations, the choice of concrete matcher models should be tailored to the underlying observation scenario, and depends on whether they are applied for static analysis (where run times usually play a secondary role) or in online contexts (where matching runs have to be determined repeatedly in limited time). Especially the raw matcher model and observation scenarios that allow for several types of imprecisions (in particular with larger deviations of multiple data variables) can result in run times up to 30 s which are only feasible in a static setting. Likewise, matching large numbers of observation points measured over long time frames will result in worse performance, which, however, can be mitigated in an online context by matching more often over shorter time frames. Overall, the matching run times lie below 1 s in most cases where the observation scope is chosen accordingly (e.g., observation sequence lengths up to 100), and thus would allow for the application of the matcher models in online contexts as well.

While we applied our matching approach to a model suite that mainly consists of demonstration examples and two medium-size case study models, the results certainly transfer to larger models as well. The additional transitions over helper locations add a constant to execution times of each original model transition, and the additional checks for clock and value bounds of imprecise observations add another constant to each transition that statically depends on the number of observed data variables. Therefore, the temporal extent and numbers of observations remain the deciding factors for performance, and the size of the model may only influence the initial generation times of the matcher system and the overall performance of state exploration in that model (which of course then applies to the checking of any property on the original model, not only our matcher properties).

## 7 Conclusion and future work

In this work, we approached the problem of observation matching on the runs of a model to check if given observations are contained in the model, and provided a solution based on model checking the reachability of an acceptance state in a synthesized matcher system. We investigated observation matching for a variety of imprecisions, including partial, deviating, qualitative, time-shifted, and committed observations, provided 4 versions of matcher model templates to cover these cases, and proved the correctness of observation matching using the appropriate template. The experiments confirmed the applicability of our approach and its implementation (as all positives and negatives were correctly classified), and, regarding performance, showed in particular that it is reasonable to choose matcher models tailored to the characteristics of observations (instead of using one template supporting all features for all cases), and that the matching time is negatively impacted by imprecision and scarcity of observation points, as increasing ambiguities and

belated discarding of non-matching runs increase the potential state space to explore.

In future work, one may investigate further types of observations and matcher models. In terms of observations, one could consider a combination of timed actions and variable data, which were considered separately in this work, and furthermore explore if other imprecisions besides the ones covered in this work are worth consideration. In terms of matcher models, one may consider restructuring the model to omit helper locations if possible, as they introduce additional intermediate states and thus increase the state space. Also, other state matching approaches may be considered as alternatives to the matching window we implemented. Finally, one may perform further experiments regarding classification of negatives, whose generation is, as we pointed out during the experiments, a challenging task in general. In a broader sense, the approach may be used in model checking to narrow down the potential futures relevant for checking of those paths branching from a given observed and matched past.

## A SBLL definition

For reference, this section provides the syntax and semantics definitions of $SBLL$ as defined in [2] with minor changes to the notation to fit our definition of states, and without the formulae $\langle a\rangle tt$ ($a \in \mathscr{U}$), $X$, and $max(X, \varphi)$, which are not needed in our work.

**Definition 3** (*SBLL syntax*) The $SBLL$ syntax is inductively defined as:

$$\begin{aligned} \varphi ::= \quad & \mathtt{ff} \quad | \quad \varphi_1 \wedge \varphi_2 \quad | g \vee \varphi \mid \Cap\varphi \mid \\ & [a]\varphi \quad | \quad x \ \underline{in}\ \varphi \\ g ::= {} & x \sim p \mid x - y \sim p \end{aligned} \tag{30}$$

where $a \in \Sigma, x, y \in K$ (clocks), $p \in \mathbb{N}$, and $\sim \in \{<, >, =\}$.

**Definition 4** (*SBLL semantics*) The $SBLL$ semantics is inductively defined as the relation $\models$ on the transition system $TS(M)$ of a timed automaton $M$ with the extended states $(l, u, v)$, where $l$ is a location in $M$, $u$ is a clock valuation, and $v$ is a variable valuation:

$$\begin{aligned} & (l,u,v) \models \mathtt{ff} \\ \Leftrightarrow\ & false \\ & (l,u,v) \models \varphi_1 \wedge \varphi_2 \\ \Leftrightarrow\ & \forall l', v' : (l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v') \\ & \quad \text{implies } (l',u,v') \models \varphi_1 \text{ and } (l',u,v') \models \varphi_2 \\ & (l,u,v) \models g \vee \varphi \\ \Leftrightarrow\ & g(u) \text{ or } \forall l', v' : (l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v') \\ & \quad \text{implies } (l',u,v') \models \varphi \\ & (l,u,v) \models \Cap\varphi \\ \Leftrightarrow\ & \forall d \in \mathbb{R}_{\geq 0} : \forall l', v' : (l,u,v) \overset{\varepsilon(d)}{\Longrightarrow} (l',u+d,v') \\ & \quad \text{implies } (l',u+d,v') \models \varphi \\ & (l,u,v) \models [a]\varphi \\ \Leftrightarrow\ & \forall l', v' : (l,u,v) \overset{a}{\Longrightarrow} (l',u,v') \\ & \quad \text{implies } (l',u,v') \models \varphi \\ & (l,u,v) \models x \ \underline{in}\ \varphi \\ \Leftrightarrow\ & \forall l', v' : (l,u,v) \xrightarrow{\tau}{}^{*} (l',[x \to 0]u,v') \\ & \quad \text{implies } (l',[x \to 0]u,v') \models \varphi \end{aligned}$$

## B Proofs

### B.1 Proof: *srl_to_sbll* returns SBLL formulae

Lemma 1 is clear for all formulae $\varphi_{SRL}$ in Eq. (21) except for the sub-formulae containing negations. For these two exceptions, each $SRL$ constraint $g = x \sim p$ and $g = x - y \sim p$, with $\sim \in \{\leq, \geq\}$, turns into $x \sim' p$ and $x - y \sim' p$ by negation, respectively, with $\sim' \in \{<, >\}$, which are syntactically defined constraints in $SBLL$. Likewise, $(g_{var})$ turns into constraints in $SBLL$ as $\sim_v$ is closed under negation. Thus, Lemma 1 holds. □

### B.2 Proof: negative equivalence established by *srl_to_sbll*

We prove Lemma 2 by case distinction and structural induction, where we assume the induction hypothesis $\neg(TS(M) \models_{SBLL} srl\_to\_sbll(\varphi) \iff (TS(M) \models_{SRL} \varphi)$. For better readability, we abbreviate $srl\_to\_sbll$ by $f$.

- Case $\psi := \mathtt{tt}$ (base case):

$$\begin{aligned} & \neg((l,u,v) \models_{SBLL} f(\mathtt{tt})) && \text{(Eq. (21))} \\ =\ & \neg((l,u,v) \models_{SBLL} \mathtt{ff}) && \text{(SBLL sem.)} \\ \Leftrightarrow\ & \neg false \\ =\ & true && \text{(SRL sem.)} \\ \Leftrightarrow\ & (l,u,v) \models_{SRL} \mathtt{tt} \end{aligned}$$

- Case $\psi := g \wedge \varphi$:

$$\begin{aligned} & \neg((l,u,v) \models_{SBLL} f(g \wedge \varphi)) && \text{(Eq. (21))} \\ =\ & \neg((l,u,v) \models_{SBLL} (\neg g \vee f(\varphi))) && \text{(SBLL sem.)} \\ \Leftrightarrow\ & \neg(\neg g(u) \text{ or } (\forall l', v' : \\ & \quad (l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v') \\ & \quad \text{implies } (l',u,v') \models_{SBLL} f(\varphi)) && \text{(De Mor.)} \\ =\ & g(u) \text{ and } \exists l', v' : \end{aligned}$$

$(l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v')$
and $\neg((l',u,v') \models_{SBLL} f(\varphi))$ (Ind. hyp.)
$\Leftrightarrow g(u)$ and $\exists l', v' :$
$(l,u,v) \xrightarrow{\tau}{}^{*} (l',u,v')$
and $(l',u,v') \models_{SRL} \varphi$ (SRL sem.)
$\Leftrightarrow (l,u,v) \models_{SRL} (g \wedge \varphi)$

- Case $\psi = g_{var} \wedge \varphi$: Apply the proof for $g \wedge \varphi$ with $g_{var}$ instead of $g$ (cf. Equation (20)).
- Case $\psi = \exists\!\!\!\!\mathbb{\exists}\varphi$:

$\neg((l,u,v) \models_{SBLL} f(\exists\!\!\!\!\mathbb{\exists}\varphi))$ (Eq. (21))
$= \neg((l,u,v) \models_{SBLL} \mathbb{\forall} f(\varphi))$ (SBLL sem.)
$\Leftrightarrow \neg(\forall d \in \mathbb{R}_{\geq 0} : \forall l', v' :$
$(l,u,v) \xRightarrow{\varepsilon(d)} (l',u+d,v')$
implies $(l',u+d,v') \models_{SBLL} f(\varphi))$
$= \exists d \in \mathbb{R}_{\geq 0} : \exists l', v' :$
$(l,u,v) \xRightarrow{\varepsilon(d)} (l',u+d,v')$
and $\neg((l',u+d,v') \models_{SBLL} f(\varphi))$ (Ind. hyp.)
$\Leftrightarrow \exists d \in \mathbb{R}_{\geq 0} : \exists l', v' :$
$(l,u,v) \xRightarrow{\varepsilon(d)} (l',u+d,v')$
and $(l',u+d,v') \models_{SRL} f(\varphi)$ (SRL sem.)
$\Leftrightarrow (l,u,v) \models_{SRL} \exists\!\!\!\!\mathbb{\exists}\varphi$

- Case $\psi = \langle a \rangle \varphi$:

$\neg((l,u,v) \models_{SBLL} f(\langle a \rangle \varphi))$ (Eq. (21))
$= \neg((l,u,v) \models_{SBLL} [a] f(\varphi))$ (SBLL sem.)
$\Leftrightarrow \neg(\forall l', v' : (l,u,v) \xRightarrow{a} (l',u,v')$
implies $(l',u,v') \models_{SBLL} f(\varphi))$
$= \exists l', v' : (l,u,v) \xRightarrow{a} (l',u,v')$
and $\neg((l',u,v') \models_{SBLL} f(\varphi))$ (Ind. hyp.)
$\Leftrightarrow \exists l', v' : (l,u,v) \xRightarrow{a} (l',u,v')$
and $(l',u,v') \models_{SRL} \varphi$ (SRL sem.)
$\Leftrightarrow (l,u,v) \models_{SRL} \langle a \rangle \varphi$

- Case $\psi = x \underline{in}_{\exists} \varphi$:

$\neg((l,u,v) \models_{SBLL} f(x \underline{in}_{\exists} \varphi))$ (Eq. (21))
$= \neg((l,u,v) \models_{SBLL} x \underline{in} f(\varphi))$ (SBLL sem.)
$\Leftrightarrow \neg(\forall l', v' : (l,u,v) \xrightarrow{\tau}{}^{*}$
$(l', [x \to 0]u, v')$ implies
$(l', [x \to 0]u, v') \models_{SBLL} f(\varphi))$
$= \exists l', v' : (l,u,v) \xrightarrow{\tau}{}^{*}$
$(l', [x \to 0]u, v')$ and
$\neg((l', [x \to 0]u, v') \models_{SBLL} f(\varphi))$ (Ind. hyp.)
$\Leftrightarrow \exists l', v' : (l,u,v) \xrightarrow{\tau}{}^{*}$
$(l', [x \to 0]u, v')$ and
$(l', [x \to 0]u, v') \models_{SRL} \varphi$ (SRL sem.)
$\Leftrightarrow (l,u,v) \models_{SRL} x \underline{in}_{\exists} \varphi$

□

### B.3 Proof: $m_T$ reachable iff $M$ models $\varphi_{SRL}$

Using [2], Eq. (21), and Lemma 2, we get:

$\neg reach(TS_{pc}, m_T)$ (via [2])
$\Leftrightarrow TS(M) \models \varphi_{SBLL}$ (subst. $\varphi_{SBLL}$)
$= TS(M) \models srl\_to\_sbll(\varphi_{SRL})$ (via Lemma 2)
$\Leftrightarrow \neg(TS(M) \models \varphi_{SRL})$.

By canceling out the negations on both sides, we get $reach(TS_{pc}, m_T) \iff (TS(M) \models \varphi_{SRL})$, and thus, Lemma 3 holds. □

### B.4 Proof: $\varphi_{part,SRL}$, $\varphi_{dev,SRL}$, and $\varphi_{loc,SRL}$ model partial, deviating, and location matching

We structure the proof similarly to the proof for simple observation sequence matching (cf. Lemma 4). For partial or deviating observations, we apply $match_{state,part}$ (cf. Equation (12)), checking that all observation variables are either $NOB$ or match the variable values of a path in the model, or $match_{state,dev}(dt, dvs)$ (cf. Equation (13)), checking that all observed time and variable values deviate at most $dt$ and $dvs$, respectively, from the variable values of a path in the model. We obtain the relations $R_{part} = \{(obs_{seq,1} = (t_1, NOB, \ldots, val_{n,1}), (t_1, val_{1,1}, \ldots, val_{n,1})), \ldots, (obs_{seq,m} = (t_m, val_{1,m}, \ldots, NOB), (t_n, val_{1,m}, \ldots, val_{n,m}))\}$ and $R_{dev} = \{(obs_{seq,1} = (t_{1,dev}, val_{1,1,dev}, \ldots, val_{n,1,dev}), (t_1, val_{1,1}, \ldots, val_{n,1})), \ldots, (obs_{seq,m} = (t_{m,dev}, val_{1,m,dev}, \ldots, val_{n,m,dev}), (t_n, val_{1,m}, \ldots, val_{n,m}))\}$. For both relations, the 4 matching criteria of Eq. (10) are fulfilled. For location observations, the proof of Lemma 4 can be directly applied by representing (non-)active locations as Boolean variables. Thus, Lemma 8 holds. □

### B.5 Proof: $\varphi_{delay,SRL}$ models time-shifted observation matching

As in Sect. B.4, confer Lemma 4 for the base proof using $match_{state,eq}$. For matching time-shifted observations, we apply $match_{state,delay}(d)$ (cf. Equation (14)) as state matcher function, which checks that all observation states match the states of a path in the model, shifting the observations altogether by $d_s \leq d$ time units. We obtain the relation $R = \{(obs_{tr,1} = (t_1 + d_s, val_{1,1}, \ldots), (t_1, val_{1,1}, \ldots)), \ldots, (obs_{tr,m} = (t_m + d_s,$

$val_{1,m}, \ldots), (t_m, val_{1,m}, \ldots))\}$, fulfilling the 4 matching criteria of Eq. (10). Thus, Lemma 9 holds. □

### B.6 Proof: preservation of runs

The initial sub-run $r_{m,init}$ in the matcher system $M_m$ has the following form:

$$\begin{aligned} r_{m,init} &= (\langle \texttt{l\_init}, u_{o,init}, v_{o,init}\rangle \\ &\quad \| \langle \texttt{I\_m}, u_{m,init}, \{(\texttt{i}, 0), (\texttt{\_stepped}, \mathit{true})\}\rangle) \\ &\xrightarrow{\tau} (\langle \texttt{l\_init}, u_{o,init}, v_{o,init}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,init}, \{(\texttt{i}, 0), (\texttt{\_stepped}, \mathit{false})\}\rangle), \end{aligned}$$

and is transformed to the sub-run $r_{o,init}$ as follows:

$$r_{o,init} = tf(r_{m,init}) = \langle \texttt{l\_init}, u_{o,init}, v_{o,init}\rangle,$$

which is simply the initial state of $M$.

The sub-run $r_{m,mc}$ in the matcher cycle of $M_m$ has the following form:

$$\begin{aligned} r_{m,mc} &= (\langle \texttt{l}, u_{o,0}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,0}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xRightarrow{\varepsilon(d)} (\langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xrightarrow{\tau} (\langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{h}, u_{m,1}, \{(\texttt{i}, i+1), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xrightarrow{\tau} (\langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, i+1), (\texttt{\_stepped}, \mathit{false})\}\rangle), \end{aligned}$$

and is transformed to the sub-run $r_{o,mc}$ as follows:

$$\begin{aligned} r_{o,mc} &= tf(r_{m,mc}) \\ &= \langle \texttt{l}, u_{o,0}, v_{o,0}\rangle \\ &\xRightarrow{\varepsilon(d)} \langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \\ &\xrightarrow{\tau} \langle \texttt{l}, u_{o,1}, v_{o,0}\rangle, \end{aligned}$$

which is an enabled delay transition in $M$, as the clock valuation $u_{o,1}$ reached by the delay $d$ and satisfying the constraints of $M$ and $T$ imposed in the parallel composition $M_m$ trivially also satisfies the constraints imposed by $M$ alone, and $\langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \xrightarrow{\tau} \langle \texttt{l}, u_{o,1}, v_{o,0}\rangle$ is a plain self-loop.

The sub-run $r_{m,sc}$ in the stepper cycle of $M_m$ has the following form:

$$\begin{aligned} r_{m,sc} &= (\langle \texttt{l\_1}, u_{o,0}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,0}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xRightarrow{\varepsilon(d)} (\langle \texttt{l\_1}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xrightarrow{\tau} (\langle \texttt{l\_int}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{true})\}\rangle) \\ &\xrightarrow{\tau} (\langle \texttt{l\_int}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{I\_m}, u_{m,1}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{true})\}\rangle) \\ &\xrightarrow{\mathit{\_step}} (\langle \texttt{l\_2}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, i+1), (\texttt{\_stepped}, \mathit{false})\}\rangle), \end{aligned}$$

and is transformed to the sub-run $r_{o,sc}$ as follows:

$$\begin{aligned} r_{o,sc} &= tf(r_{m,sc}) \\ &= \langle \texttt{l\_1}, u_{o,0}, v_{o,0}\rangle \\ &\xRightarrow{\varepsilon(d)} \langle \texttt{l\_1}, u_{o,1}, v_{o,0}\rangle \\ &\xrightarrow{\tau} \langle \texttt{l\_2}, u_{o,1}, v_{o,0}\rangle, \end{aligned}$$

which is an enabled sequence of a delay transition and an action transition in $M$. The delay transition is enabled as explained earlier for $r_{o,mc}$. The action transition is enabled as the order of locations of $M$ remains unchanged in the parallel composition $M_m$, and the conditions required for the transition from `l_1` to `l_2` in $M$ (i.e., potential clock guards, conditions on variables, and enabled actions) were not affected by $T$ in $M_m$. Thus, if the action transition was enabled under the additional synchronization constraints imposed by `_stepped` and `_step` in $M_m$, it is certainly enabled without them in $M$.

Finally, the sub-run $r_{m,end}$ in $M_m$ has the following form:

$$\begin{aligned} r_{m,end} &= (\langle \texttt{l}, u_{o,0}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,0}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xRightarrow{\varepsilon(d)} (\langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_i}, u_{m,1}, \{(\texttt{i}, i), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xrightarrow{\tau} (\langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{h}, u_{m,1}, \{(\texttt{i}, i+1), (\texttt{\_stepped}, \mathit{false})\}\rangle) \\ &\xrightarrow{\tau} (\langle \texttt{l}, u_{o,1}, v_{o,0}\rangle \\ &\quad \| \langle \texttt{m\_t}, u_{m,1}, \{(\texttt{i}, i+1), (\texttt{\_stepped}, \mathit{false})\}\rangle), \end{aligned}$$

and is transformed to the sub-run $r_{o,end}$ as follows:

$$\begin{aligned} r_{o,end} &= tf(r_{m,end}) \\ &= \langle \texttt{l}, u_{o,0}, v_{o,0}\rangle \\ &\xRightarrow{\varepsilon(d)} \langle \texttt{l}, u_{o,1}, v_{o,0}\rangle, \end{aligned}$$

which again is an enabled delay transition in $M$ (cf. the explanation for $r_{o,mc}$).

In summary, each transformed run $r_o = tf(r_m) = r_{o,init} + \{r_{o,mc}, r_{o,sc}\}^* + r_{o,end}$ starts in the initial state of $M$, performs

a sequence of delay and action transitions enabled in $M$, and ends with a final delay transition again enabled in $M$. Thus, we conclude that Lemma 10 holds. □

**Acknowledgements** We thank the anonymous reviewers many times who provided a great amount of helpful comments that improved the readability and accessibility of the article.

**Funding** Open Access funding enabled and organized by Projekt DEAL. Part of this research has been funded by the TUHH i$^3$ lab initiative (T-LP-E01-WTM-1801-02) and DFG/SCHU 2479.

**Data availability** The models `2doors`, `bridge`, `fischer`, `fischer-symmetry`, `interrupt`, `train-gate`, and `train-gate-orig` are part of the demo model suite of Uppaal available at http://www.uppaal.org/. The models `csmacd2` [35] and `tdma` [36] were developed in case studies. All other experimental data were programmatically generated by the authors via the experiments linked under *Code availability*.

**Code availability** The project implementation is open source under the MIT license. The Uppaal observation matcher is available at https://github.com/S-Lehmann/uppyyl-observation-matcher, and the examples and experiments covered in this article are altogether available at https://github.com/S-Lehmann/uppyyl-observation-matcher-experiments.

## Declarations

**Conflict of interest** Not applicable.



## References

1. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: a practical approach. Formal Aspects Comput. **20**(4), 481–505 (2008). https://doi.org/10.1007/s00165-008-0082-7
2. Aceto, L., Bouyer, P., Burgueño, A., Larsen, K.G.: The power of reachability testing for timed automata. Theoret. Comput. Sci. **300**(1), 411–475 (2003). https://doi.org/10.1016/S0304-3975(02)00334-1
3. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal 4.0 (2006)
4. Havelund, K., Larsen, K., Skou, A.: Formal verification of a power controller using the real-time model checker UPPAAL. In: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems. ARTS '99, vol. 1601, pp. 277–298 (1999). https://doi.org/10.1007/3-540-48778-6_17
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the Second Workshop on Formal Methods in Software Practice. FMSP '98, pp. 7–15 (1998). https://doi.org/10.1145/298595.298598
6. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. IEEE Trans. Softw. Eng. **41**(7), 620–638 (2015). https://doi.org/10.1109/TSE.2015.2398877
7. André, É.: Observer patterns for real-time systems. In: 2013 18th International Conference on Engineering of Complex Computer Systems, pp. 125–134 (2013). https://doi.org/10.1109/ICECCS.2013.26
8. Abid, N., Dal Zilio, S., Botlan, D.: Real-time specification patterns and tools (2013). https://doi.org/10.1007/978-3-642-32469-7_1
9. Konrad, S., Cheng, B.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering. ICSE '05, pp. 372–381 (2005). https://doi.org/10.1109/ICSE.2005.1553580
10. Vogel, T., Carwehl, M., Rodrigues, G.N., Grunske, L.: A property specification pattern catalog for real-time system verification with UPPAAL. Inf. Softw. Technol. **154**, 107100 (2023). https://doi.org/10.1016/j.infsof.2022.107100
11. Ulus, D., Ferrère, T., Asarin, E., Maler, O.: Timed pattern matching. In: Formal Modeling and Analysis of Timed Systems, pp. 222–236 (2014). https://doi.org/10.1007/978-3-319-10512-3_16
12. Bakhirkin, A., Ferrère, T., Nickovic, D., Maler, O., Asarin, E.: Online timed pattern matching using automata. In: Formal Modeling and Analysis of Timed Systems, pp. 215–232 (2018). https://doi.org/10.1007/978-3-030-00151-3_13
13. Waga, M., André, E., Hasuo, I.: Parametric timed pattern matching. ACM Trans. Softw. Eng. Methodol. **32**(1), 1–35 (2022). https://doi.org/10.1145/3517194
14. Alur, R., Kurshan, R.P., Viswanathan, M.: Membership questions for timed and hybrid automata. In: Proceedings 19th IEEE Real-Time Systems Symposium, pp. 254–263 (1998). https://doi.org/10.1109/REAL.1998.739751
15. Uhl, T.: The inverse identification problem and its technical application. Arch. Appl. Mech. **77**(5), 325–337 (2007). https://doi.org/10.1007/s00419-006-0086-9
16. Bourke, T.P.: Modelling and programming embedded controllers with timed automata and synchronous languages. PhD Thesis, University of New South Wales (2009)
17. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S., Probst, D.: Property preserving abstractions for the verification of concurrent systems. Formal Methods Syst. Des. **6**(1), 11–44 (1995). https://doi.org/10.1007/BF01384313
18. Namjoshi, K.S.: Abstraction for branching time properties. In: Computer Aided Verification, pp. 288–300 (2003). https://doi.org/10.1007/978-3-540-45069-6_29
19. Dyck, J., Giese, H., Lambers, L.: Automatic verification of behavior preservation at the transformation level for relational model transformation. Softw. Syst. Model. **18**(5), 2937–2972 (2019). https://doi.org/10.1007/s10270-018-00706-9
20. Braunstein, C., Encrenaz, E.: CTL-property transformations along an incremental design process. Int. J. Softw. Tools Technol. Transf. **9**(1), 77–88 (2007). https://doi.org/10.1007/s10009-006-0007-9
21. Henzinger, T.A., Majumdar, R., Prabhu, V.S.: Quantifying similarities between timed systems. In: Formal Modeling and Analysis of Timed Systems, pp. 226–241 (2005). https://doi.org/10.1007/11603009_18
22. Lin, C.-K., Fan, K.-C., Tze-Pa Lee, F.: On-line recognition by deviation-expansion model and dynamic programming matching.

Pattern Recogn. **26**(2), 259–268 (1993). https://doi.org/10.1016/0031-3203(93)90034-T
23. Milani, A., Jassó, J., Suriani, S.: Modeling online user behavior. In: 2008 IEEE International Conference on e-Business Engineering, pp. 736–741 (2008). https://doi.org/10.1109/ICEBE.2008.113
24. Krichen, M., Tripakis, S.: Black-box conformance testing for real-time systems. In: Model Checking Software, pp. 109–126 (2004). https://doi.org/10.1007/978-3-540-24732-6_8
25. Gilbert, D., Hogger, C., Zlatuska, J.: Transforming specifications of observable behaviour into programs. In: Logic Program Synthesis and Transformation—Meta-Programming in Logic, pp. 88–103 (1994). https://doi.org/10.1007/3-540-58792-6_6
26. Dragomir, I., Iosti, S., Bozga, M., Bensalem, S.: Designing systems with detection and reconfiguration capabilities: A formal approach. In: Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems, pp. 155–171 (2018). https://doi.org/10.1007/978-3-030-03424-5_11
27. Agrawal, M., Stephan, F., Thiagarajan, P.S., Yang, S.: Behavioural approximations for restricted linear differential hybrid automata. In: Hybrid Systems: Computation and Control, pp. 4–18 (2006).https://doi.org/10.1007/11730637_4
28. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers, pp. 77–117 (2008). https://doi.org/10.1007/978-3-540-78917-8_3
29. David, A., Larsen, K., Legay, A., Nyman, U., Wasowski, A.: ECDAR: An environment for compositional design and analysis of real time systems. In: Automated Technology for Verification and Analysis: 8th International Symposium, pp. 365–370 (2010). https://doi.org/10.1007/978-3-642-15643-4_29
30. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: Runtime Verification, pp. 165–184 (2019). https://doi.org/10.1007/978-3-030-32079-9_10
31. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL—a tool suite for automatic verification of real-time systems. In: Hybrid Systems III, pp. 232–243 (1996)
32. Lehmann, S.: Uppaal Observation Matcher. https://github.com/S-Lehmann/uppyyl-observation-matcher (2023)
33. Lehmann, S.: Uppaal Observation Matcher Experiments. https://github.com/S-Lehmann/uppyyl-observation-matcher-experiments (2023)
34. Bengtsson, J.: Clocks, DBMs and states in timed systems. PhD thesis, Uppsala University (2002)
35. Jensen, H., Larsen, K., Skou, A.: Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. BRICS Rep. Ser. **3**(24) (1996). https://doi.org/10.7146/brics.v3i24.20005
36. Lonn, H., Pettersson, P.: Formal verification of a TDMA protocol start-up mechanism. In: Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems, pp. 235–242 (1997). https://doi.org/10.1109/PRFTS.1997.640153



**Sascha Lehmann** received his M.Sc. in computer science and engineering from the Hamburg University of Technology (TUHH). In the following years, he worked as a research assistant at the Institute for Software Systems (TUHH), where he engaged in teaching (in the fields of functional programming and software testing) and research, with a focus on the application of formal methods in online contexts and their potential use in medical applications such as the steering of flexible needles in soft tissue.

**Sibylle Schupp** received her Ph.D. in computer science from the University of Tuebingen, Germany. Following, she spent one year as postdoctoral researcher at Rensselaer in upstate New York and subsequently accepted an offer for an Assistant Professorship, also at Rensselaer. In the Summer of 2002 Dr. Schupp moved to Gothenburg, Sweden, to join the Department of Computer Science and Engineering at Chalmers University of Technology (CTH) as Associate Professor. Since 2009, Dr. Schupp is a full professor for computer science at Hamburg University of Technology (TUHH) and head of the Institute for Software Systems. Sibylle Schupp's interests center around rigorous methods for software quality assurance, often in safety-critical contexts, which allow for formal verification of system properties.