Science of Computer Programming 77 (2012) 779–798

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

SAT-based verification for timed component connectors

S. Kemper

Centrum Wiskunde en Informatica, Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 30 November 2009
Received in revised form 3 February 2011
Accepted 8 February 2011
Available online 24 February 2011

Keywords:
Timed constraint automata
Abstraction refinement
Model checking
SAT
Component-based software engineering

ABSTRACT

Component-based software construction relies on suitable models underlying components, and in particular the coordinators which orchestrate component behaviour. Verifying correctness and safety of such systems amounts to model checking the underlying system model. The model checking techniques not only need to be correct (since system sizes increase), but also scalable and efficient.

In this paper, we present a SAT-based approach for bounded model checking of Timed Constraint Automata, which permits true concurrency in the timed orchestration of components. We present an embedding of bounded model checking into propositional logic with linear arithmetic. We define a product that is linear in the size of the system, and in this way overcome the state explosion problem to deal with larger systems. To further improve model checking performance, we show how to embed our approach into an extension of counterexample guided abstraction refinement with Craig interpolants.

1. Introduction

Component-based software engineering supports constructing large systems by composing individual components. The correctness and safety of these concurrent systems depend on *inter-component* communication actions which happen at the *right time*. Components are often only available as black boxes; therefore, there is a need for component connectors that provide exogenous coordination, i.e., coordination from *without* [2]. As such, these component connectors require true concurrency in time, which combines synchrony and asynchrony, to express complex coordination patters.

The computational complexity introduced by the infinite state space of these *real-time systems* leads to severe limitations in scalability even within very well-established model checkers like UPPAAL (http://www.uppaal.com). Aside from the omnipresent state explosion problem [14] already present in finite state model checking, current model checking techniques for real-time systems are still limited in the number of concurrent quantitative temporal observations (measured by clocks). A particularly dramatic cause of the state explosion problem is the exponential blow-up obtained by forming the cross product for parallel composition. To avoid this, we define a linear-size parallel composition for the logical representation of TCA. By providing an initial valuation for step 0, typically only a reduced part of the full parallel composition has to be expanded from our representation during satisfiability checking (SAT solving).

Very sophisticated and well-optimised techniques (e.g., [28]) guide high-end SAT solvers to explore only a comparably narrow fragment around the part of the state space relevant for the particular safety property. We build upon this development by choosing a linear arithmetic/propositional encoding, a philosophy that has successfully proven its great potential in finite state systems [13]. With this basis, we exploit the particularities of transition systems induced by TCA using abstraction refinement to deal with the challenges of infinite states.

E-mail address: s.kemper@cwi.nl.

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.02.003

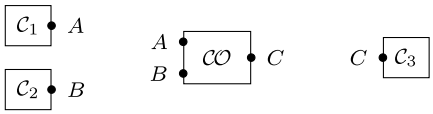

Fig. 1. TCA, conceptual view.

1.1. Timed constraint automata

Timed Constraint Automata [4] (TCA) are a combination of constraint automata [5] (CA) and timed automata (TA) with location invariants [1]. Originally defined as a semantical model for the channel based coordination language $\mathcal{R}$eo [3] (and so far having been used for this single purpose only), they offer a powerful coordination mechanism for channel based coordination languages in general. In this work, we exploit the full modelling spectrum of the formalism, by *directly* using TCA to implement coordinating connectors in networks where timed components communicate by exchanging data through multiple channels. The behaviour of the network is given by synchronisation between channel ends (ports).

While the functionality of channels is often limited to synchrony and (FIFO) buffering, TCA allow connectors with arbitrary behaviour. These connectors provide exogenous coordination, by imposing a certain communication pattern – for example reordering or delays – on associated components. TCA are compositional, which allows to easily build complex connectors out of simpler ones.

Most action-based (coordination) models, like e.g. finite state machines, I/O automata or TA, permit only a single action per transition. As a consequence, synchrony, and concurrent execution of actions in the parallel composition, is reduced to arbitrary interleavings plus nondeterminism. Especially for timed systems (like TA) – aside from being unintuitive – this does not correctly capture the nature of distributed systems, since it imposes a sequential order on actions which conceptually happen at the same time. Moreover, from a technical point of view, the presence of all possible interleavings amplifies the state explosion problem. In contrast, TCA allow sets of actions on each transition, which permits *true concurrency*, as this directly models (truly atomic) synchronous communication through different ports.

In this way, TCA provide a coordination model to implement component connectors, which combines the notions of real-time and true concurrency, and allows for complex coordination patterns including both synchrony and asynchrony.

Example 1.1 (*Introduction*)**.** Fig. 1 shows our conceptual notion of networks of TCA. The network presented here consists of three components $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$, and a central connector $\mathcal{CO}$, which is connected to the components through ports A, B and C, respectively. Throughout the paper, we will use the connector $\mathcal{CO}$ as a running (toy) example. The basic idea of $\mathcal{CO}$ is to repeatedly receive data from components $\mathcal{C}_1$ (through port A) and $\mathcal{C}_2$ (through B), and to send this data to $\mathcal{C}_3$ (through C). Depending on constraints on the received data, either component $\mathcal{C}_1$ or $\mathcal{C}_2$ is connected to – i.e., data is transmitted between – component $\mathcal{C}_3$ (dynamic reconfiguration). Note that this is just a toy example, used to illustrate the concepts introduced in this paper. For a more meaningful example, consider Section 6.

1.2. Abstraction refinement

Abstraction refinement [14,19] is a promising direction of research to overcome the challenges of the state explosion problem and infinite state model checking, while preserving correctness of verification results. Abstraction techniques based on over-approximation (also called *conservative* approximation) enrich the system behaviour, by removing constraints that are considered irrelevant for verifying a particular property. These techniques may yield false negatives: a safety property is rejected in the abstract system, though it holds in the original system. If, however, the abstract system is safe (no error state is reachable) then, by over-approximation, so is the original.

Based on the representation of TCA in propositional logic with linear arithmetic, iterative abstraction refinement consists of the following steps: applying the abstraction function to the representation, we automatically produce a simpler *abstract* version of it. After *unfolding* the resulting transition formula k times, a *satisfiability check* solves the bounded reachability question in the abstract system. Depending on the outcome, the system has either been proven safe (error state is unreachable) within bound k, or needs to be analysed with respect to an abstract counterexample (*concretised*), again using SAT solving. If the abstract counterexample has a counterpart in the non-abstracted system, then the system is unsafe. Otherwise, the counterexample is *spurious* and results from an inappropriate choice of abstraction. Analysing the counterexample (with Craig interpolants derived by the SAT solver, e.g. FOCI (based on [25]) or MathSAT [24]) then helps to *refine* the abstraction and start over until the system is proven safe (within bound k) or unsafe.

1.3. Contributions

The main contributions of this paper can be summarised as follows: we extend the (expressiveness of the) basic definition of TCA [4], by generalising from finite data domains to countably infinite data domains.

We then develop a formal framework for direct investigation and verification of TCA: we define a constraint-based representation of TCA in propositional logic with linear arithmetic. These constraints capture the current state of connectors in the network, and possible synchronisations of connectors with each other and the environment. In this way, we can

benefit from existing SAT and constraint solving techniques when model checking safety and correctness properties of the connectors.

Based on the representation of TCA, we define an abstraction function, which removes constraints that are considered irrelevant to a particular property, and which in this way reduces the state explosion problem and increases the manageable system size.

We finally illustrate the approach by means of a case study.

Implementation. Most parts of the framework presented here are already publicly available as plug-in for the Eclipse Coordination Tools (ECT, [16]): we have extended the CA editor to support editing of TCA (including various syntactical checks, e.g. well-formedness of clock constraints). Within this platform, we have implemented the translation of TCA to propositional formulae with linear arithmetic constraints (front end), as described in this paper. Further, we have implemented the generation of input files for the MathSAT solver (back end), including support for products of TCA, allowing us to analyse the underlying TCA in detail. Having split the formula generation in two parts, it is very easy to switch to another solver, by just exchanging the back end. The remaining parts, in particular interactive abstraction refinement, are scheduled to be part of future releases of the ECT. See Section 6 for more details on tool support.

Organisation of the paper. In the next section, we discuss some related work. After introducing TCA and bounded model checking (BMC) in Section 2, we present a faithful representation of TCA in propositional logic with linear arithmetic for BMC in Section 3, and prove soundness and completeness of the representation in Section 4. In Section 5, we introduce a uniform abstraction, extend the algebraic perspective on soundness from Section 3 to correspondence results about abstraction, and briefly recall how to exploit spurious counterexamples for refining abstractions. In Section 6, we discuss an example in more detail, and point out available tool support. Finally, Section 7 concludes the paper and discusses some future work.

1.4. Related work

Our model of simultaneous execution of an arbitrary number of actions comes closest to Pratt's higher dimensional automata (HDA, [29]). There, n-dimensional transitions (with sets of n actions) represent the truly concurrent execution of n independent events; yet HDA contain all mutually exclusive executions (concurrent execution of a subset of actions) of these events as well.

The basic idea of TCA is equivalent to Pratt's approach: we model true concurrency by allowing sets of actions on each transition. We further extend the model in three ways. First, we add synchronisation, which is straightforward once true concurrency is established. On execution of a transition, matching actions (here: channel ends/ports connected together) are said to synchronise, while independent actions still execute concurrently as before. Second, we add the notion of real-time to the model, to allow for complex real-time coordination patterns. As a third extension, we do not require the mutually exclusive executions to be present, and in this way permit to model systems which contain only true concurrency. Furthermore, our approach is compositional, which eases modelling of complex connectors.

There are a number of process algebras and languages which permit true concurrency, for example Esterel [9], SCCS [27], or real-time agents [12]. Yet, none of them provides the combination of features which our approach supports. The former two only handle discrete time, and do not support asynchronous systems. In addition, Esterel only allows for deterministic reactive systems. Real-time agents support both synchrony and asynchrony, but synchronisation is binary, by matching complementary actions. Moreover, communication is restricted to pure synchronisation, i.e., without passage of data values.

Most standard process algebras provide extensions to support real-time, e.g. CCS, CSP or ACP [32,30,8], where ACP is the most general with respect to communication [7]. Yet, it does not allow for true concurrency. Synchronisation is realised by a user-defined binary synchronisation function, and the simultaneous occurrence of n actions is modelled by arbitrary interleaving (i.e., $n-1$ applications of the synchronisation function) plus nondeterminism, again imposing a sequential order on the execution of concurrent actions. In contrast to the aforementioned, in our framework, we combine all these features, and provide a simple, direct and easily understandable way of modelling nondeterministic interactive systems, which permit true concurrency of an arbitrary number of actions, and allow both synchronous and asynchronous communication, with or without value passing, in real-time.

Abstraction refinement approaches have been proposed by for example Jhala and McMillan [20], and by Clarke et al. [14]. In [20], the authors use interpolants to generate refinements, while taking into account specific characteristics of the property to be checked. A limitation, however, is the fact that they rely on an appropriate initial choice of predicates for predicate abstraction. Our approach can be considered as a quick (hence, scalable) approximation of predicate abstraction, where predicate discovery is evident by exploiting the nature of TCA. The approach in [14] works with Kripke structures originating from finite state programs. In contrast, our approach deals with the challenges of infinite state model checking, as introduced by the notion of real-time clocks. Further, we directly use a formula representation which is tailored for SAT-based bounded model checking.

In this paper,[1] we extend the SAT-based approach for TA presented by Kemper and Platzer [22] in a number of ways. We take into account the special transition characteristics of TCA, in particular the truly atomic execution of actions which

[1] Being an extended version of [21], we have extended the data domain of TCA (from finite to) countably infinite domains, added correctness proofs for TCA product, representation and abstraction, a more detailed case study, improved the abstraction function, and have shown how to include certain infinite behaviour in the model checking process.

happen at the same time. We extend the formula representation, by adding data constraints and handling of concrete data values. We extend the simple, yet powerful abstraction function from [22] with a more refined handling of the different syntactic categories in the formula. Our abstraction function is then able to preserve more information in the abstract case than the corresponding abstraction function in [22] (while reasoning about the same level of abstraction), which reduces the number of spurious counterexamples.

The model checker Vereofy (http://www.vereofy.de) provides tools for model checking (untimed) CA, but to the best of our knowledge, the framework presented in this work is the first approach for model checking TCA.

2. Timed constraint automata

In this section, we introduce the standard notations for TCA [4] in the time domain $\mathtt{Time}=\mathbb{R}_{\geq 0}$, and for BMC [13,10], and we model our running example.

2.1. Syntax

In what follows, let $\mathcal{P}$ be a finite, nonempty set of ports, through which TCA exchange data values, and let $\mathcal{D}ata$ be a (possibly infinite but) countable set of data values which can be sent or received via ports. For simplicity of representation, we assume a special element $\perp\in\mathcal{D}ata$ representing "no data".

Definition 2.1 (*Data Constraint, Clock Constraint*)**.** A *data assignment* $\delta\in DA(\mathcal{P})$ *over (data domain* $\mathcal{D}ata$ *and) port set* $\mathcal{P}$ is a mapping $\delta:\mathcal{P}\to\mathcal{D}ata$, assigning to each port $A\in\mathcal{P}$ the currently pending data value. If no data is pending, $\delta(A)$ evaluates to the special value "no data". We may write d_A for $\delta(A)$. A *clock valuation* $\nu\in\mathcal{V}(\mathcal{X})$ *over a set of clocks* $\mathcal{X}$ is a mapping $\nu:\mathcal{X}\to\mathtt{Time}$, assigning to each clock $x\in\mathcal{X}$ its current value. *Data constraints* $dc\in DC(\mathcal{P})$ *over (*$\mathcal{D}ata$ *and)* $\mathcal{P}$, and *clock constraints* $cc\in CC(\mathcal{X})$ *over* $\mathcal{X}$ are defined as follows:

$$dc ::= \mathtt{true} \mid d_A{=}d \mid d_A{=}d_B \mid dc_1\wedge dc_2 \mid \neg dc, \text{ with } A, B\in\mathcal{P} \text{ and } d\in\mathcal{D}ata$$
$$cc ::= \mathtt{true} \mid x\sim n \mid cc_1\wedge cc_2, \text{with } x\in\mathcal{X},\ n\in\mathbb{N} \text{ and } \sim\in\{<,\leq,=,\geq,>\}.$$

Other data constraints, for example $d_A\in D$, $D\subseteq\mathcal{D}ata$, $dc_1\vee dc_2$, or $dc_1\to dc_2$, are defined as abbreviations ("syntactic sugar") in the standard way. We use $\models$ for the standard satisfaction relation. For example, $\nu\models(x\sim c)$ iff $\nu(x)\sim c$.

Note that we assume clock constraints to be *convex*, i.e., they do not contain $\vee$, $\neg$ [1]. Intuitively, convexity of a clock constraint cc means that for any two clock valuations ν and ν', with $\nu(x)<\nu'(x)$ for some clock x, if $\nu\models cc$ and $\nu'\models cc$, then $\nu''\models cc$ for all ν'' with $\nu(x)<\nu''(x)<\nu'(x)$. This property is used for efficient representation. Non-convex clock constraints however can be simulated by splitting locations (for invariants) or transitions (for guards).

Definition 2.2 (*Timed Constraint Automaton*)**.** A *TCA (over* $\mathcal{D}ata$*)* is a tuple $\mathcal{T}=(S,\mathcal{X},\mathcal{P},E,s_0,I)$, with S a finite set of locations, $s_0\in S$ the initial location, $\mathcal{X}$ a finite set of clocks, $\mathcal{P}$ a finite set of ports, $I:S\to CC(\mathcal{X})$ a function assigning a clock constraint (*location invariant*) to every location, and $E\subseteq S\times 2^{\mathcal{P}}\times DC(\mathcal{P})\times CC(\mathcal{X})\times 2^{\mathcal{X}}\times S$ the finite set of transitions. For every transition $e=(s,P,dc,cc,X,s')\in E$, we require $dc\in DC(P)$ (*data guard of* e) and $cc\in CC(\mathcal{X})$ (*clock guard of* e), and both satisfiable. X is the *clock set of* e, P the *port set of* e; if $P=\emptyset$, e is called *invisible*, otherwise, it is called *visible*.

The idea of invisible transitions is that they do not represent observable data flow (since $P=\emptyset$, note that $dc=\mathtt{true}$ in this case, since we require $dc\in DC(P)$), but just serve for internal synchronisation purposes, for example by resetting clocks. Visible transitions, on the other hand, correspond to observable behaviour: an element $e=(s,P,dc,cc,X,s')\in E$ describes a transition from location s to location s', where data flows through all ports in the port set P. After the TCA has delayed in location s for a positive amount of time (during which the invariant $I(s)$ of s needs to be satisfied), it may execute the transition and move to location s', provided that the data values pending at ports in P satisfy the data guard dc, and the clock values satisfy the clock guard cc and the invariant $I(s')$ of the target location s'. The firing of the transition, i.e., the location change from s to s', is considered to be instantaneous. On execution of the transition, all clocks in the clock set X are reset to zero. The timing constraints will be made explicit in the definition of semantics (Definition 2.8).

We do not impose any semantic constraints on data and clock guards. Clock guards may for example "overlap", like $x\leq 4\wedge x\geq 3$. Yet, due to convexity, this simply reduces the number of satisfying valuations. Moreover, guards do not need to be complete, i.e., cover every possible valuation. This may lead to so-called *timelocks* [2] [31], but by definition, such behaviour is excluded from the semantics (see Definition 2.8 and explanations thereafter).

We now introduce the TCA for our running example.

Example 2.3 (*TCA*)**.** Fig. 2 shows the TCA of the connector $\mathcal{CO}$ (left, cf. Example 1.1) and a simple component $\mathcal{C}$ (right), which may communicate through port A. We assume $\mathcal{D}ata=\{1,2\}$ (thus, actually $\mathcal{D}ata=\{1,2,\perp\}$), and we omit constraints equal to $\mathtt{true}$ as well as empty sets on transitions.

[2] As an example of a timelock, consider a location with invariant $x\leq 3$ and a single outgoing transition with clock guard $x\leq 2$. The location cannot be left once $\nu(x)>2$, and as soon as $\nu(x)=3$, time cannot progress anymore, since the automaton is neither allowed stay in the location nor allowed to leave it.

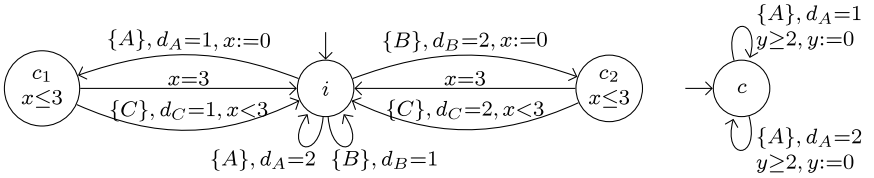

Fig. 2. TCA Example.

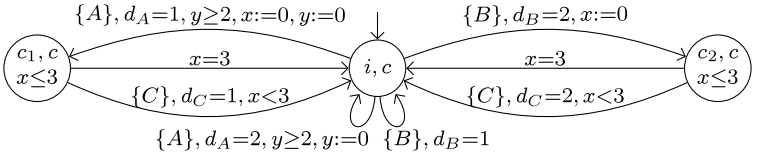

Fig. 3. TCA Example: Product.

The intended behaviour of $\mathcal{CO}$ is as follows: it accepts data value 1 through port *A*, and data value 2 through port *B*. All other data values are discarded, represented by the two self loops in location *i* ("idle"). When receiving data value 1 through *A* (upper left transition), $\mathcal{CO}$ moves to location c_1, and resets its clock *x*. This is done to force an upper time bound on the delivery of data values through *C*. If in location c_1, the data value can be transmitted through *C* before clock *x* has reached the threshold, $\mathcal{CO}$ moves back to the initial location (lower left transition), where it is ready to accept the next input. Otherwise – i.e., the data value cannot be transmitted before three time units have elapsed – the invariant ($x \leq 3$) of location c_1 forces $\mathcal{CO}$ to leave c_1 and to move back to location *i* (centre left transition), without transmitting the data value. The explanation for c_2 is analogous.

The behaviour of component $\mathcal{C}$ is simple: it repeatedly sends data items through port *A*, with a minimum delay of 2 time units between subsequent transmissions. Note that in the graphical representation, we use assignment rather than set notation for the clocks contained in the clock set.

Remark 2.4 (*Notation of TCA*)**.** If not stated otherwise, we shall assume the constituents of a TCA $\mathcal{T}$ to be denoted as $\mathcal{T}=(S, \mathfrak{X}, \mathcal{P}, E, s_0, I)$, and of a TCA $\mathcal{T}_i$ to be denoted as $\mathcal{T}_i=(S_i, \mathfrak{X}_i, \mathcal{P}_i, E_i, s_{0,i}, I_i)$, for $i \in \mathbb{N}$.

Within a system of TCA, two automata synchronise – i.e., communicate – if the port sets of the involved transitions coincide on common ports. This gives rise to the following definition.

Definition 2.5 (*Product of TCA*)**.** Let $\mathcal{T}_i$ be TCA over $\mathcal{D}ata_i$, $i=1, 2$, with $\mathfrak{X}_1 \cap \mathfrak{X}_2 = \emptyset$ and $S_1 \cap S_2 = \emptyset$ (can be achieved by renaming the constituents in one of the TCA). The *product of* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ is a new TCA $\mathcal{T}_1 \bowtie \mathcal{T}_2 = (S, \mathfrak{X}, \mathcal{P}, E, s_0, I)$ over $\mathcal{D}ata_1 \cup \mathcal{D}ata_2$, with $S = S_1 \times S_2$, $\mathfrak{X} = \mathfrak{X}_1 \cup \mathfrak{X}_2$, $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$, $I: S_1 \times S_2 \to CC(\mathfrak{X}_1 \cup \mathfrak{X}_2)$ such that $I(s_1, s_2) = I_1(s_1) \wedge I_2(s_2)$, $s_0 = (s_{0,1}, s_{0,2})$, and E is defined by

$$\frac{\begin{array}{c}(s_1, P_1, dc_1, cc_1, X_1, s_1') \in E_1 \\ (s_2, P_2, dc_2, cc_2, X_2, s_2') \in E_2 \\ P_1 \cap \mathcal{P}_2 = P_2 \cap \mathcal{P}_1,\ P_1 \neq \emptyset,\ P_2 \neq \emptyset,\ dc_1 \wedge dc_2 \neq \texttt{false}\end{array}}{(\langle s_1, s_2 \rangle, P_1 \cup P_2, dc_1 \wedge dc_2, cc_1 \wedge cc_2, X_1 \cup X_2, \langle s_1', s_2' \rangle) \in E} \tag{1}$$

$$\frac{(s_1, P_1, dc_1, cc_1, X_1, s_1') \in E_1,\ P_1 \cap \mathcal{P}_2 = \emptyset,\ s_2 \in S_2}{(\langle s_1, s_2 \rangle, P_1, dc_1, cc_1, X_1, \langle s_1', s_2 \rangle) \in E} \tag{2}$$

and the symmetric rule of the latter.

Rule (1) captures the synchronisation of visible transitions: the nonempty port sets have to coincide on common ports, i.e. data flows through the same set of shared ports on both transitions. The case where $P_1 \cap \mathcal{P}_2 = P_2 \cap \mathcal{P}_1 = \emptyset$ (i.e., the set of shared ports is empty) represents a system step where each automaton performs a *local* visible transition, (concurrent execution of independent actions). Rule (2) describes the execution of a local transition (visible or invisible) in one automaton, while the other automaton remains in its current location. Note that in case this local transition in the first automaton is preceded by a time delay, the second automaton actually performs a delay transition. An example for the product construction can be found in Fig. 3: it shows the product TCA $\mathcal{CO} \bowtie \mathcal{C}$, for the TCA presented in Example 2.3.

Proposition 2.6 (*Product of TCA*)**.** *The product of TCA is commutative and associative, up to isomorphy of location names.*

Proof. 1. Commutativity follows from the commutativity of $\cup$ on port and clock sets, and the commutativity of $\wedge$ on data and clock constraints.

2. Associativity follows from the associativity of $\cup$ on port and clock sets, the associativity of $\wedge$ on data and clock constraints, and the fact that for $P_i \subseteq \mathcal{P}_i$, $i=1, 2, 3$, if $P_2 \cap \mathcal{P}_3 = P_3 \cap \mathcal{P}_2$, $P_1 \cap (\mathcal{P}_2 \cup \mathcal{P}_3) = \mathcal{P}_1 \cap (P_2 \cup P_3)$ and $P_1 \cap \mathcal{P}_2 = P_2 \cap \mathcal{P}_1$, then $P_3 \cap (\mathcal{P}_1 \cup \mathcal{P}_2) = \mathcal{P}_3 \cap (P_1 \cup P_2)$.

It may be required to hide some ports of a TCA from the environment. For example, in the product construction, the common ports could be considered to become "internal" ports, and thus not be visible from the outside any more. The hiding operation (cf. [4]) removes all information about a set of ports $O \subseteq \mathcal{P}$ from a TCA. To ensure correct timed behaviour of transitions with port sets $P \subseteq O$ – namely that such transitions may only be taken after a positive amount of time – we need to introduce an additional clock.

Definition 2.7 (*Hiding in TCA*)**.** Let $\mathcal{T}$ be a TCA, $x \notin \mathcal{X}$ a clock, and $O \subseteq \mathcal{P}$. The *hiding of port set O in $\mathcal{T}$* yields a new TCA $\mathcal{T}\backslash_O = (S, \mathcal{X} \cup x, \mathcal{P} \backslash O, E', s_0, I)$, where E' is given by

$$\frac{(s, P, dc, cc, X, s') \in E,\ ((P = \emptyset) \vee (P \backslash O \neq \emptyset))}{(s, P \backslash O, dc\backslash_O, cc, X \cup x, s') \in E} \tag{3}$$

$$\frac{(s, P, dc, cc, X, s') \in E,\ \emptyset \neq P \subseteq O}{(s, \emptyset, \mathtt{true}, cc \wedge (x > 0), X \cup x, s') \in E'} \tag{4}$$

Here, $dc\backslash_O$ denotes the data constraint which is derived from dc by replacing all literals $(d_A = d)$, $\neg(d_A = d)$, $(d_A = d_B)$ and $\neg(d_A = d_B)$ (cf. Definition 2.1) by $\mathtt{true}$ for all $A \in O$.

The basic idea is to obtain transitions of $\mathcal{T}\backslash_O$ from transitions of $\mathcal{T}$ by reducing the port set (and data constraint) to ports not contained in O (3). If the resulting transition in $\mathcal{T}\backslash_O$ is invisible, while the underlying transition in $\mathcal{T}$ is visible (4), the new clock x is used to ensure correct timed behaviour: since x is reset on all transitions, the additional constraint $(x > 0)$ ensures the elapse of a positive amount of time before the (now invisible) transition can be taken.

2.2. Semantics

TCA model true concurrency, by allowing sets of ports on each transition. As a consequence, a positive amount of time has to elapse before every visible transition (while invisible transitions may be instantaneous). The underlying idea is that all actions which happen at the same time are truly atomic and thus collapse to a single transition. The semantics of TCA is defined as the set of runs of the associated labelled transition system (LTS) $\mathfrak{S}_\mathcal{T}$ [4].

Definition 2.8 (*Associated LTS*)**.** Let $\mathcal{T}$ be a TCA. The *associated LTS* $\mathfrak{S}_\mathcal{T}$ is a tuple $\mathfrak{S}_\mathcal{T} = (\mathcal{Q}, q_0, \rightarrow)$, with $\mathcal{Q} \subseteq (S \times \mathcal{V}(\mathcal{X}))$ the set of configurations, such that $\nu \models I(s)$ for every $(s, \nu) \in \mathcal{Q}$, $q_0 = \langle s_0, \mathbf{0} \rangle$ the initial configuration, with $\mathbf{0}(x) = 0$ for all $x \in \mathcal{X}$, and the transition relation $\rightarrow \subseteq \mathcal{Q} \times 2^\mathcal{P} \times DA(\mathcal{P}) \times \mathtt{Time} \times \mathcal{Q}$ is given by

$$\frac{\begin{array}{l}(s, P, dc, cc, X, s') \in E \\ t > 0,\ t \geq t' \geq 0 : \nu + t' \models I(s) \\ (\nu + t)[X := 0] \models I(s'),\ \nu + t \models cc \\ \delta \in DA(P) : \delta \models dc\end{array}}{\langle s, \nu \rangle \xrightarrow{P, \delta, t} \langle s', \nu + t[X := 0] \rangle} \tag{5}$$

$$\frac{\begin{array}{l}(s, \emptyset, \mathtt{true}, cc, X, s') \in E \\ \nu[X := 0] \models I(s') \\ \nu \models cc\end{array}}{\langle s, \nu \rangle, \xrightarrow{\emptyset, \emptyset, 0} \langle s', \nu[X := 0] \rangle} \tag{6}$$

Valuation $\nu + t$ (*timeshift*) increases all clocks by the same amount of time t, and valuation $\nu[X := 0]$ (*modification*) resets the values of all clocks $x \in X$ to zero.

Rule (5) captures the constraints described after Definition 2.2, for both visible and invisible transitions: before the transition can be fired, a positive amount of time $(t > 0)$ has to elapse, during which the invariant $I(s)$ needs to be satisfied (second row).[3] At time t, the transition is fired and resets all clocks in the clock set to zero, provided that the clock guard is satisfied *before* the resetting of clocks, and the invariant of the target location is satisfied *after* the resetting of clocks (third row). Moreover, the data values pending at the ports (given by the data assignment δ) have to satisfy the data guard (fourth row). Rule (6) captures the fact that invisible transitions may be instantaneous; it can be seen as a simplification of (5) for $P = \emptyset$ and $t = 0$.

A *run of $\mathfrak{S}_\mathcal{T}$ starting in configuration q*, denoted by $\mathbf{r}$, is a sequence of transitions $\mathbf{r} = q \xrightarrow{P, \delta, t} q_1 \xrightarrow{P', \delta', t'} \ldots$ which is either time divergent (i.e. infinite, and $t + t' + \cdots = \infty$) or finite and ends in a *terminal configuration* $\langle s, \nu \rangle$ (i.e. without outgoing transitions, allowing for infinite passage of time: $\forall t > 0 : \nu + t \models I(s)$). The *trace semantics of $\mathcal{T}$* is given by the set $Run_\mathcal{T}$ of initial

[3] Due to convexity, these constraints can be relaxed in the representation, since it is enough to check the invariant at the beginning and at the end of the time delay.

runs (i.e., starting in the initial configuration) of $\mathfrak{S}_{\mathcal{T}}$. With $Run_{\mathcal{T},k}$, we denote the set of finite prefixes of elements of $Run_{\mathcal{T}}$ of (at most) length k. In (7), we show a run of the product $\mathcal{CO}\bowtie\mathcal{C}$ (cf. Fig. 3) of length 5.

$$\begin{aligned}\left\langle\langle i,c\rangle,\left[\begin{smallmatrix}x=0\\y=0\end{smallmatrix}\right]\right\rangle &\xrightarrow{\{A\},d_A=2,2.5}\left\langle\langle i,c\rangle,\left[\begin{smallmatrix}x=2.5\\y=0\end{smallmatrix}\right]\right\rangle\xrightarrow{\{A\},d_A=1,2}\left\langle\langle c_1,c\rangle,\left[\begin{smallmatrix}x=0\\y=0\end{smallmatrix}\right]\right\rangle\xrightarrow{\{C\},d_C=1,1}\\ &\left\langle\langle i,c\rangle,\left[\begin{smallmatrix}x=1\\y=1\end{smallmatrix}\right]\right\rangle\xrightarrow{\{B\},d_B=2,1}\left\langle\langle c_2,c\rangle,\left[\begin{smallmatrix}x=0\\y=2\end{smallmatrix}\right]\right\rangle\xrightarrow{\emptyset,\emptyset,3}\left\langle\langle i,c\rangle,\left[\begin{smallmatrix}x=3\\y=5\end{smallmatrix}\right]\right\rangle\end{aligned}\tag{7}$$

2.3. Bounded model checking

Bounded model checking (BMC) has turned out to be amongst the most promising approaches for verification of safety properties [13,10]. These properties declare what should not happen – or equivalently, what should always happen – and are typically expressed as reachability properties. Safety properties can be disproved with a finite counterexample, i.e., a finite run, where the last configuration contains a contradiction to the property. The principle of BMC for safety properties is to examine prefix fragments of the transition system, and successively increase the exploration bound until it reaches (a computable indicator of) the diameter of the system – in which case the system has been proven safe – or an unsafe run has been discovered [6].

Definition 2.9 (*Bounded Safety*)**.** Let $\mathcal{T}$ be a TCA, $s\in S$ an error location. $\mathcal{T}$ is *safe with respect to s within bound k*, denoted by $\mathcal{T}\models_k\neg\exists\Diamond s$, if there is no run in $Run_{\mathcal{T},k}$ containing s. Otherwise, $\mathcal{T}$ is *unsafe with respect to s*.

The lifting of $\neg\exists\Diamond s$ to reason about configurations rather than locations is straightforward. On the basis of these reachability properties, other bounded LTL specifications can be verified as well, using the encoding in [6].

3. Representation of timed constraint automata

In this section, we construct a formula $\varphi(\mathcal{T})$ in propositional logic with linear arithmetic that represents the behaviour of a TCA $\mathcal{T}$ (given by the runs of $\mathfrak{S}_{\mathcal{T}}$, cf. Section 2.2), by defining transition characteristics from step $\mathtt{t}\text{-}1$ to step $\mathtt{t}$, $\mathtt{t}\in\mathbb{N}_{>0}$ (Section 3.2). For BMC, we unfold $\varphi(\mathcal{T})$ k times (for $\mathtt{k}$ steps), which yields a formula $\varphi(\mathcal{T})_k$ representing all (prefixes of) runs of $\mathfrak{S}_{\mathcal{T}}$ for $\mathtt{k}$ steps, and we show how to extend this finite unfolding to infinite, ultimately periodic runs (Section 3.3). The formula $\varphi(\mathcal{T})_k$, together with a representation of the safety property, is unsatisfiable iff $\mathcal{T}$ is safe within bound k (cf. Definition 2.9). Finally, we give a representation of the product of two TCA which is *linear* in the size of the automata (Section 3.4), and we show how the notion of hiding is defined on the formula representation (Section 3.5).

3.1. Basic components

The possible behaviour of a TCA depends on the values of its constituents (clocks, locations, data pending at ports), and changes over time. Therefore, we "parametrise" the variables representing these constituents by the step $\mathtt{t}$ they are evaluated in, and we call this *localisation*: the localisation ψ_t of a formula ψ is obtained by adding index t to all variable symbols occurring in ψ. Thus, if ψ is of vocabulary x, s, d, then ψ_t is of vocabulary $\mathtt{x_t}$, $\mathtt{s_t}$, $\mathtt{d_t}$ instead. In particular, we use:

Locations For every location $s\in S$, the Boolean variable $\mathtt{s_t}$ represents whether the TCA is in location s in step $\mathtt{t}$.

Data values, ports The injective mapping $\Delta{:}\mathcal{D}ata\to\mathbb{N}$ maps each $d_i\in\mathcal{D}ata$ to a natural number $\mathtt{n^i}$ (the *representation of* d_i), with $0\stackrel{\text{def}}{=}\mathtt{n^0}\stackrel{\text{def}}{=}\mathtt{n^\perp}$ representing $\perp$. For every port $A\in\mathcal{P}$, the Boolean *activity variable* $\mathtt{A_t}$ *of* A represents whether data flows through A in step t, and the natural *data variable* $\mathtt{DA_t}$ *of* A represent which data occurs at A in step $\mathtt{t}$. In case of no data flow, $\mathtt{DA_t}$ evaluates to $\mathtt{n^\perp}$.

Data constraints For a data constraint $dc = (d_A{=}d_i)$ (cf. Definition 2.1), with $\Delta(d_i){=}\mathtt{n^i}$, the formula $\mathtt{A_t}\wedge\mathtt{dc_t}$, with $\mathtt{dc_t} = (\mathtt{DA_t}{=}\mathtt{n^i})$, evaluates to $\mathtt{true}$ iff $\delta(A){=}d_i$ in step $\mathtt{t}$. For a data constraint $dc = (d_A{=}d_B)$, the formula $\mathtt{A_t}\wedge\mathtt{B_t}\wedge\mathtt{dc_t}$, with $\mathtt{dc_t} = (\mathtt{DA_t}{=}\mathtt{DB_t})$, evaluates to $\mathtt{true}$ iff $d_A{=}d_B$ in step $\mathtt{t}$.

Clocks For every clock $x\in\mathcal{X}$, the rational variable $\mathtt{x_t}$ (*clock reference*) represents the absolute point in time where x was last reset prior to step $\mathtt{t}$. An additional rational variable $\mathtt{z_t}$ (*absolute time reference*) represents the absolute amount of time that has passed until step $\mathtt{t}$. The clock value of x at step $\mathtt{t}$ is thus obtained by $\mathtt{z_t}-\mathtt{x_t}$. Note that linear arithmetic is equisatisfiable for rational and real variables [22].

Clock constraints For a clock constraint $cc = x\sim n$ (cf. Definition 2.1), the formula $\mathtt{z_t}-\mathtt{x_t}\sim n$, denoted as $\mathtt{cc_t}$, evaluates to $\mathtt{true}$ iff cc holds in step $\mathtt{t}$, and the formula $\mathtt{z_t}-\mathtt{x_{t\text{-}1}}\sim n$ (*inter-step representation*), denoted as $\mathtt{cc_{t\Delta}}$, evaluates to $\mathtt{true}$ iff cc holds in step $\mathtt{t}$ and x has not been reset since step $\mathtt{t}\text{-}1$.

The representation of other constraints is straightforward, using conjunctions (and negations, for data constraints) of the aforementioned representations.

The inter-step representation is needed for correct representation of delayed transitions in $\mathfrak{S}_{\mathcal{T}}$ (5), i.e. transitions which are preceded by a positive amount of time: the invariant of the target location s' is evaluated under the valuation $\nu{+}t[X{:=}0]$, that means *after* the time delay and *after* the execution of the transition. In contrast, the invariant of the source location s and the clock guard of the transition are evaluated under the valuation $\nu{+}t$, that means *after* the passage of time, but *before* the execution of the transition. The inter-step representation is used to access the clock value at this particular point in time "in the middle" of the execution step. See Example 3.3 for more details.

$$\varphi^{init}(\mathcal{T}) = \bar{\mathtt{s}}_0 \wedge \mathtt{I}(\bar{\mathtt{s}})_0 \wedge \bigwedge_{s\in S, s\neq\bar{s}} \neg \mathtt{s}_0 \wedge \bigwedge_{A\in\mathcal{P}} (\neg \mathtt{A}_0 \wedge (\mathtt{DA}_0 = \mathtt{n}^{\perp})) \wedge \bigwedge_{x\in X} (\mathtt{x}_0 = 0) \wedge (\mathtt{z}_0 = 0) \tag{8}$$

$$\varphi^{visible}(e) = \mathtt{s}_{t\text{-}1} \wedge \bigwedge_{A\in P} \mathtt{A}_t \wedge \bigwedge_{A\notin P} \neg \mathtt{A}_t \wedge \mathtt{dc}_t \wedge \bigwedge_{x\in X} (\mathtt{x}_t = \mathtt{z}_t) \wedge \bigwedge_{x\notin X} (\mathtt{x}_t = \mathtt{x}_{t\text{-}1}) \wedge (\mathtt{z}_{t\text{-}1} < \mathtt{z}_t) \wedge \mathtt{cc}_{t\Delta} \wedge \mathtt{I}(\mathtt{s})_{t\Delta} \wedge \mathtt{s'}_t \wedge \mathtt{I}(\mathtt{s'})_t \tag{9}$$

$$\varphi^{invisible}(e') = \mathtt{s}_{t\text{-}1} \wedge \bigwedge_{A\in\mathcal{P}} \neg \mathtt{A}_t \wedge \bigwedge_{x\in X} (\mathtt{x}_t = \mathtt{z}_t) \wedge \bigwedge_{x\notin X} (\mathtt{x}_t = \mathtt{x}_{t\text{-}1}) \wedge (\mathtt{z}_{t\text{-}1} \leq \mathtt{z}_t) \wedge \mathtt{cc}_{t\Delta} \wedge \mathtt{I}(\mathtt{s})_{t\Delta} \wedge \mathtt{s'}_t \wedge \mathtt{I}(\mathtt{s'})_t \tag{10}$$

$$\varphi^{trans}(\mathcal{T}) = \bigvee_{e\in E, P\neq\emptyset} \varphi^{visible}(e) \vee \bigvee_{e'\in E, P=\emptyset} \varphi^{invisible}(e') \tag{11}$$

$$\varphi^{location}(\mathcal{T}) = \bigvee_{s\in S} (\mathtt{s}_t \wedge \bigwedge_{s'\in S, s'\neq s} \neg \mathtt{s}'_t) \tag{12}$$

$$\varphi^{data\_value}(\mathcal{T}) = \bigwedge_{A\in\mathcal{P}} (\neg \mathtt{A}_t \vee \neg(\mathtt{DA}_t = \mathtt{n}^{\perp})) \wedge (\mathtt{A}_t \vee (\mathtt{DA}_t = \mathtt{n}^{\perp})) \tag{13}$$

$$\varphi(\mathcal{T}) = \varphi^{init}(\mathcal{T}) \wedge \varphi^{trans}(\mathcal{T}) \wedge \varphi^{location}(\mathcal{T}) \wedge \varphi^{data\_value}(\mathcal{T}) \tag{14}$$

Fig. 4. Transition relation representation.

3.2. Transition relation

The representation of the transition relation needs to take care of the special behaviour of TCA, namely, that every visible transition has to be preceded by a positive time delay, whereas invisible transitions may be instantaneous. It constrains the possible valuations of variables representing the configuration at subsequent step $\mathtt{t}$ depending on those at step $\mathtt{t\text{-}1}$. Conceptually, the delay is represented by evolving from $\mathtt{t\text{-}1}$ to $\mathtt{t}$, while the (instantaneous) location change takes place at $\mathtt{t}$.

Definition 3.1 (*Timed Constraint Automaton Representation*)**.** Let $\mathcal{T}$ be a TCA, let $e=(s, P, dc, cc, X, s')$ and $e'=(s, \emptyset, \mathtt{true}, cc, X, s')$ be a visible and invisible transition in $\mathcal{T}$, respectively. The *formula representation* $\varphi(\mathcal{T})$ *of the transition relation of* $\mathcal{T}$ is defined in (14) in Fig. 4.

The automaton starts in its initial location $\bar{s}$ (8) in step $\mathtt{0}$ (to avoid confusion with localisation indices, we denote the initial location as $\bar{s}$ rather than s_0, so its representation is $\bar{\mathtt{s}}_0$ rather than the odd-looking $(\mathtt{s}_0)_0$), the invariant of which has to be satisfied, data must not flow through any port, and all clocks are set to zero. Before executing a visible transition (9) in step $\mathtt{t}$, $\mathcal{T}$ is in location s. After the elapse of a positive amount of time ($\mathtt{z}_{t\text{-}1} < \mathtt{z}_t$), after which the invariant $\mathtt{I}(\mathtt{s})_{t\Delta}$ of s and the clock guard $\mathtt{cc}_{t\Delta}$ of the transition hold, $\mathcal{T}$ switches to location s', the invariant of which has to hold. All clocks referenced in the clock set X are set to the actual point in time, while the values of the other clocks do not change. Data flows through all ports A contained in the port set P, while the other ports are inactive, and the data constraint $\mathtt{dc}_t$ is satisfied. Due to convexity, the invariant needs to be checked at the end of the time delay only, as it inductively holds at the beginning (8). The execution of an invisible transition (10) is similar, except that the amount of time elapsed may be zero, and data must not flow through any port. The disjunction of all visible and invisible transitions expresses nondeterministic transition choice (11). In any step, the current location is unique (12), and the special value "no data" may only be pending at inactive ports (13).

Remark 3.2 (*Finite Data Domain*)**.** If $|\mathcal{D}ata|=k$ (finite), we require the elements of $\mathcal{D}ata$ to be mapped to subsequent natural numbers (remember that $\Delta(\perp)=0$, and Δ is injective), such that $\Delta(\mathcal{D}ata)=\{0, \ldots, k-1\} \subset \mathbb{N}$, and we add a constraint $\bigwedge_{A\in\mathcal{P}}(\mathtt{DA}_t \leq k-1)$ to φ^{data_value} (13). This speeds up verification, since the number of possible valuations (for data variables) decreases.

Example 3.3 (*Representation of TCA*)**.** Consider again the TCA $\mathcal{C}$ in Fig. 2. With Δ such that $\Delta(\perp)=0$, $\Delta(1)=1$, and $\Delta(2)=2$, the representation of $\mathcal{C}$ according to Definition 3.1 is given in (15) (we omit constraints equal to $\mathtt{true}$).

$$\begin{aligned}
\varphi^{init}(\mathcal{C}) &= \mathtt{c}_0 \wedge \neg \mathtt{A}_0 \wedge (\mathtt{DA}_0 = 0) \wedge (\mathtt{y}_0 = 0) \wedge (\mathtt{z}_0 = 0) \\
\varphi^{visible}(\mathcal{C}) &= (\mathtt{c}_{t\text{-}1} \wedge \mathtt{A}_t \wedge (\mathtt{DA}_t = 1) \wedge (\mathtt{y}_t = \mathtt{z}_t) \wedge (\mathtt{z}_{t\text{-}1} < \mathtt{z}_t) \wedge (\mathtt{z}_t - \mathtt{y}_{t\text{-}1} \geq 2) \wedge \mathtt{c}_t) \vee \\
&\quad (\mathtt{c}_{t\text{-}1} \wedge \mathtt{A}_t \wedge (\mathtt{DA}_t = 2) \wedge (\mathtt{y}_t = \mathtt{z}_t) \wedge (\mathtt{z}_{t\text{-}1} < \mathtt{z}_t) \wedge (\mathtt{z}_t - \mathtt{y}_{t\text{-}1} \geq 2) \wedge \mathtt{c}_t) \\
\varphi^{trans}(\mathcal{C}) &= \varphi^{visible}(\mathcal{C}) \\
\varphi^{data_value}(\mathcal{C}) &= (\mathtt{DA}_t \leq 2) \wedge (\neg \mathtt{A}_t \vee \neg(\mathtt{DA}_t = 0)) \wedge (\mathtt{A}_t \vee (\mathtt{DA}_t = 0))) \\
\varphi(\mathcal{C}) &= \varphi^{init}(\mathcal{C}) \wedge \varphi^{trans}(\mathcal{C}) \wedge \varphi^{data_value}(\mathcal{C})
\end{aligned} \tag{15}$$

3.3. Unfolding for bounded model checking

In order to represent the reachability problem of BMC for a TCA $\mathcal{T}$ in logic, the formula representation $\varphi(\mathcal{T})$ (14) is unfolded, i.e., instantiated for all steps up to bound k. The resulting formula $\varphi(\mathcal{T})_k$ is called *k-unfolding of $\mathcal{T}$*, and is defined in (16), where $\psi_{j/t}$ denotes the localisation (cf. Section 3.1) of ψ, with index t replaced by j.

$$\varphi(\mathcal{T})_k = \bigwedge_{1\le j\le k} \varphi(\mathcal{T})_{j/t} \tag{16}$$

Intuitively, a satisfying interpretation (called *model*) σ of $\varphi(\mathcal{T})_k$ corresponds to a run of $\mathfrak{S}_{\mathcal{T}}$ of length k, i.e., to one possible behaviour of $\mathcal{T}$ for the first k steps. Consequently, the set $\mathcal{V}(\varphi(\mathcal{T})_k)$ of all models of $\varphi(\mathcal{T})_k$ describes all possible behaviours of $\mathcal{T}$ for the first k steps. Moreover, certain models of $\varphi(\mathcal{T})_k$ not only correspond to finite runs, but to *infinite, ultimately periodic runs*. An infinite, ultimately periodic run is a run of the form (cf. Definition 2.8)

$$q_0 \xrightarrow{P_0,\delta_0,t_0} \cdots \xrightarrow{P,\delta,t} q_l \xrightarrow{P_l,\delta_l,t_l} \cdots \xrightarrow{P_{l'},\delta_{l'},t_{l'}} q_l \xrightarrow{P_l,\delta_l,t_l} \cdots \xrightarrow{P_{l'},\delta_{l'},t_{l'}} q_l \cdots$$

that means a run which, after some finite prefix (q_0 to q_l), executes the loop from q_l back to q_l infinitely often. Conceptually, such an infinite, ultimately periodic run can be seen as a run of length k

$$q_0 \xrightarrow{P_0,\delta_0,t_0} \cdots \xrightarrow{P,\delta,t} q_l \xrightarrow{P_l,\delta_l,t_l} \cdots \xrightarrow{P_{l'},\delta_{l'},t_{l'}} q_k,$$

where the last configuration q_k is equal to configuration q_l ($l<k$). The cycle condition (17) expresses this (cf. [6]): it requires the variables constituting the configurations (i.e., locations and clock values, cf. Sections 2.2 and 3.1) to be equal in steps l and k. A model of $\varphi(\mathcal{T})_k$, which in addition satisfies the cycle condition, thus corresponds to an infinite, ultimately periodic run.

$$\varphi_k^C(\mathcal{T}) = \bigvee_{0\le l<k} \Big(\bigwedge_{s\in S} (\mathtt{s}_k{=}\mathtt{s}_1) \wedge \bigwedge_{x\in\mathcal{X}} ((\mathtt{z}_k{-}\mathtt{x}_k){=}(\mathtt{z}_1{-}\mathtt{x}_1)) \Big) \tag{17}$$

$$\varphi(\mathcal{T})_{k,C} = \varphi_k^C(\mathcal{T}) \wedge \varphi(\mathcal{T})_k \tag{18}$$

The conjunction of k-unfolding and cycle condition (18) allows us to reason about infinite, ultimately periodic runs only,[4] while (16) includes both infinite, ultimately periodic runs and finite runs of length k.

In what follows, we shall reason about $\varphi(\mathcal{T})_k$ only. Yet, since every model of $\varphi(\mathcal{T})_{k,C}$ is also a model of $\varphi(\mathcal{T})_k$, all results are valid for $\varphi(\mathcal{T})_{k,C}$ as well.

Checking the reachability of an error location s amounts to conjoining $\varphi(\mathcal{T})_k$ with the representation $\rho \stackrel{\text{def}}{=} \mathtt{s}_0 \vee \mathtt{s}_1 \vee \cdots \vee \mathtt{s}_k$ of the reachability property, such that $\mathcal{T} \models_k \neg\exists\Diamond s$ holds iff the conjunction $\varphi(\mathcal{T})_k \wedge \rho$ is unsatisfiable. Lifting ρ to reason about configurations or even execution sequences is straightforward. For example, an LTL property $s \rightarrow \bigcirc s'$ ("if the current location is s, then the next location will be s'") can be represented as $\rho = (\mathtt{s}_0 \wedge \mathtt{s}'_1) \vee (\mathtt{s}_1 \wedge \mathtt{s}'_2) \vee \cdots (\mathtt{s}_{k-1} \wedge \mathtt{s}'_k)$.

3.4. Product of timed constraint automata

The cross product of TCA, as defined in Definition 2.5, is exponential in the worst case, which is a severe limitation to the size of systems that can be verified. Here, we define a logical representation of systems of TCA which is *linear* in the number of automata. The basic idea is to retain the representations of the individual automata, and model check their juxtaposition. We require variables representing common ports to have the same name in both representations, such that constraints involving these ports are automatically satisfied simultaneously in both representation.

To model single local transitions, as described by (2) in Definition 2.5, we introduce explicit delay transitions (cf. Section 2.1): the representation of a delay transition $\varphi^{delay}(s)$ in location s is defined in (19). Note that these delay transitions are in accordance with Definition 2.2, as they correspond to invisible loops of the form $(s, \emptyset, \mathtt{true}, \mathtt{true}, \emptyset, s)$. Therefore, in particular, (19) permits zero-delays. For two TCA $\mathcal{T}_1$ and $\mathcal{T}_2$, with $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$ and $S_1 \cap S_2 = \emptyset$ (can be achieved by renaming the constituents in one of the TCA), the *representation of $\mathcal{T}_1 \bowtie\!\lhd \mathcal{T}_2$*, denoted as $\varphi(\mathcal{T}_1 \bowtie\!\lhd \mathcal{T}_2)$, is given in (20), where (11) is understood to be the disjunction of (9), (10) and (19). The *k-unfolding of the product* is defined in the same way as for individual automata, it is shown in (21).

$$\varphi^{delay}(s) = \mathtt{s}_{t-1} \wedge \bigwedge_{A\in\mathcal{P}} \neg\mathtt{A}_t \wedge \bigwedge_{x\in\mathcal{X}} (\mathtt{x}_t{=}\mathtt{x}_{t-1}) \wedge (\mathtt{z}_{t-1}{\le}\mathtt{z}_t) \wedge \mathtt{I(s)}_{t\Delta} \wedge \mathtt{s}_t \wedge \mathtt{I(s)}_t \tag{19}$$

$$\varphi(\mathcal{T}_1 \bowtie\!\lhd \mathcal{T}_2) = \varphi(\mathcal{T}_1) \wedge \varphi(\mathcal{T}_2) \tag{20}$$

$$\varphi(\mathcal{T}_1 \bowtie\!\lhd \mathcal{T}_2)_k = \bigwedge_{1\le j\le k} \varphi(\mathcal{T}_1 \bowtie\!\lhd \mathcal{T}_2)_{j/t} \tag{21}$$

[4] Yet, every model of (18) still corresponds to a finite run of length k, namely, the prefix of the infinite, ultimately periodic run which stops after reaching configuration q_l for the second time.

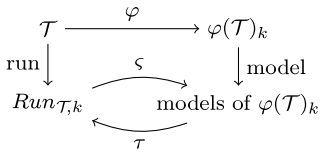

Fig. 5. Correctness of representation.

Note that the existence of such a linear product is not immediately clear, but in fact is a result of our design decision of explicitly mentioning all ports on each transition (cf. (9), (10) and (19)). This decision – though seeming unnecessary at first glance – together with the assumption that common ports have the same name, ensures that transitions in different TCA may only be executed in parallel if they fulfil the conditions described in Definition 2.5. In this way, we do not need to mention all possible synchronisations (which are allowed by (1) and (2)) explicitly, and thus avoid the exponential blow-up.

3.5. Hiding

The hiding operation removes all information about a set of ports O from a TCA $\mathcal{T}$, cf. Definition 2.7. Hiding a set of ports O in the formula representation $\varphi(\mathcal{T})$ amounts to existential quantification over the corresponding variables, i.e., activity and data variables of the ports in O, cf. Section 3.1. For a TCA $\mathcal{T}$, with formula representation $\varphi(\mathcal{T})$, and a port set $O \subseteq \mathcal{P}$, the formula representation $\varphi(\mathcal{T}\backslash_O)$ of automaton $\mathcal{T}\backslash_O$ corresponds to

$$\exists \mathtt{O}\, \varphi(\mathcal{T}), \tag{22}$$

with $\mathtt{O} = \bigcup_{A \in O, 0 \le t \le k} \{\mathtt{A}_t, \mathtt{DA}_t\}$

In Definition 2.7, an additional clock is introduced to ensure correct timed behaviour of invisible transitions in $\mathcal{T}\backslash_O$ which originate from visible transitions in $\mathcal{T}$. Here, we do not need to introduce an additional clock: the formula representation of a visible transition explicitly requires a positive amount of time to elapse ($(\mathtt{z}_{t-1} < \mathtt{z}_t)$, cf. (9)). Since $\mathtt{O}$ does not contain clock variables, this constraint remains unchanged even in case the transition becomes invisible, and therefore, correct timed behaviour, as required by Definition 2.7, is guaranteed.

3.6. Discussion

In this section, we have established a framework for exhaustive BMC of TCA, and we have shown that the unfolding not only covers finite runs (for BMC), but also infinite, ultimately periodic runs.

Using propositional formulae as intermediate representation ("front end"), we may fall back on the abstraction refinement framework of [22] ("back end"), and, more importantly, we can take advantage of existing high-performance SAT solving technologies. Our representation is specifically tailored for SAT solving: (13) is in conjunctive normal form (CNF) with binary clauses only, and (8), (9) and (10) almost entirely consist of unit clauses. With respect to speed of verification, this is very efficient: the 2-SAT problem is polynomial, and formulae with n unit clauses can be solved in $O(n)$.[5] Due to the disjunctive nature of transition choices, (11) is not in CNF, but it could easily be transformed to short CNF (see e.g. [18]) when introducing new symbols.

The restriction to convex clock constraints does not reduce the expressiveness of our model (cf. Section 2.1), but on the contrary significantly simplifies the representation formulae, since clock constraints need to be checked at the beginning and at the end of a time delay only, rather than at all intermediate points (cf. (9) and (10)). We further simplify verification by defining a product representation which is linear in the number of automata (20). In this way, we also avoid the exponential state space blow-up when forming the cross product.

Though communication is often regarded as a one-to-one relation, our representation is already suited for general n-ary communication: by having ports carrying the same name in more than two automata, our approach naturally generalises to one-to-many or even many-to-many communication models.

In the next section, we prove that our formula representation is correct and complete with respect to the underlying TCA.

4. Correctness of the representation

For the representation $\varphi(\mathcal{T})$ to be faithful (i.e., exhibit the same behaviour as $\mathcal{T}$), every model of $\varphi(\mathcal{T})_k$ has to correspond to a run of length k, and vice versa. To prove this, we show that the diagram in Fig. 5 commutes.

The commutative property expresses that models of $\varphi(\mathcal{T})_k$ have a bijective correspondence to runs of the original TCA $\mathcal{T}$, denoted by the maps ς and τ: the run $\tau(\varsigma(r))$ of the model $\varsigma(r)$ belonging to a run r again is r, and the model $\varsigma(\tau(\sigma))$ of a run $\tau(\sigma)$ belonging to a model σ again is σ.

[5] Note that (12) could be expressed in CNF with binary clauses as well. Yet, it is not, but is rather tailored for abstraction already: after abstraction, the formula in CNF would loose the information of mutual exclusion, and result in a too coarse abstraction.

Remark 4.1 (*Notation*). In what follows, we use the notation of representation variables introduced in Section 3.1, and we use the symbol $\sim$ to refer to any arithmetic comparison (cf. Definition 2.1).

For a TCA $\mathcal{T}$, we use the symbols $\mathfrak{S}_\mathcal{T}$, r and $Run_{\mathcal{T},k}$ to refer to the associated LTS, a run of $\mathfrak{S}_\mathcal{T}$, and the set of all runs of $\mathfrak{S}_\mathcal{T}$ up to length k (cf. Section 2.2). Further, we use the symbols $\varphi(\mathcal{T})_k$, σ and $\mathcal{V}(\varphi(\mathcal{T})_k)$ to refer to its k-unfolding, a model of $\varphi(\mathcal{T})_k$, and the set of all models of $\varphi(\mathcal{T})_k$, respectively (cf. Section 3.3).

We first show that the formula representation is sound, i.e., that every model $\sigma\in\mathcal{V}(\varphi(\mathcal{T})_k)$ yields a run $r\in Run_{\mathcal{T},k}$.

Definition 4.2 (*Derived Run*). For $\sigma\in\mathcal{V}(\varphi(\mathcal{T})_k)$, the *derived run* r_σ is

$$r_\sigma=\langle l_0,\nu_0\rangle\xrightarrow{P_1,\delta_1,t_1}\langle l_1,\nu_1\rangle\xrightarrow{P_2,\delta_2,t_2}\dots\xrightarrow{P_k,\delta_k,t_k}\langle l_k,\nu_k\rangle, \text{ with}$$

$$l_{k_i}=s, \text{ iff } \sigma(\mathtt{s}_{\mathtt{k}_i})=\mathtt{tt},\ ^{6} \tag{i}$$
$$\nu_{k_i}(x)=\sigma(\mathtt{z}_{\mathtt{k}_i})-\sigma(\mathtt{x}_{\mathtt{k}_i}), \tag{ii}$$
$$P_{k'+1}=\bigcup_{\sigma(\mathtt{A}_{\mathtt{k}'+1})=\mathtt{tt}}\mathtt{A}_{\mathtt{k}'+1}, \tag{iii}$$
$$\delta_{k'+1}(A)=\Delta^{-1}(\mathtt{n}^{\mathtt{i}}), \text{ iff } \sigma(\mathtt{DA}_{\mathtt{k}'+1})=\mathtt{n}^{\mathtt{i}}\neq\mathtt{n}^{\perp},\ ^{7} \text{ and} \tag{iv}$$
$$t_{k'+1}=\sigma(\mathtt{z}_{\mathtt{k}'+1})-\sigma(\mathtt{z}_{\mathtt{k}'}) \tag{v}$$

for all $0\leq k_i\leq k$, $0\leq k'<k$. The derived run for products of TCA is

$$r_\sigma:\langle(l_{0,1},l_{0,2}),\nu_0\rangle\xrightarrow{P_1,\delta_1,t_1}\dots\xrightarrow{P_k,\delta_k,t_k}\langle(l_{k,1},l_{k,2}),\nu_k\rangle,$$

which is defined in the same way, except for rewriting (i) to

$$l_{k',i}=s,\qquad \text{if } \sigma(\mathtt{s}_{\mathtt{k}'})=\mathtt{tt} \text{ and } s\in S_i,\ i=1,2, \tag{i'}$$

Lemma 4.3 (*Soundness*). *For $\sigma\in\mathcal{V}(\varphi(\mathcal{T})_k)$, the derived run r_σ is a run of $\mathfrak{S}_\mathcal{T}$ of length k, i.e., $r_\sigma\in Run_{\mathcal{T},k}$.*

Proof. Induction on k.

IA $\sigma\models\varphi(T)_0$: $\sigma(\bar{\mathtt{s}}_0)\overset{(8)}{=}\mathtt{tt}$ for the initial location $\bar{s}$, thus $l_0\overset{(i)}{=}\bar{s}$. For all clocks x, $\nu_0(x)\overset{(ii)}{=}\sigma(\mathtt{z}_0)-\sigma(\mathtt{x}_0)\overset{(8)}{=}0$, and thus $r_\sigma=\langle\bar{s},\mathbf{0}\rangle\in Run_{\mathcal{T},0}$.

IH $\sigma\models\varphi(\mathcal{T})_k$: $r_\sigma\in Run_{\mathcal{T},k}$ for $k\geq 0$.

IS $\sigma\models\varphi(\mathcal{T})_{k+1}$: $r_\sigma=\langle l_0,\nu_0\rangle\xrightarrow{P_1,\delta_1,t_1}\dots\xrightarrow{P_k,\delta_k,t_k}\langle l_k,\nu_k\rangle\xrightarrow{P_{k+1},\delta_{k+1},t_{k+1}}\langle l_{k+1},\nu_{k+1}\rangle$, and either $\sigma\models\varphi^{visible}(e)_{k+1/t}$ or $\sigma\models\varphi^{invisible}(e)_{k+1/t}$ (cf. (16)) for some e.

Case $\sigma\models\varphi^{visible}(e)_{k+1/t}$ $(*)$: let $e=(s,P,dc,cc,X,s')$ (cf. (9)), then

- $l_k\overset{\text{IH}}{=}s$, $l_{k+1}\overset{(i)}{=}s'$, $t_{k+1}\overset{(v)}{=}\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{z}_k)$.
- $\nu_{k+1}=\nu_k+t_{k+1}[X:=0]$: we have
  - for $x\in X$: $\nu_{k+1}(x)\overset{(ii)}{=}\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{x}_{k+1})\overset{(*)}{=}\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{z}_{k+1})=0$
  - for $x\notin X$: $\nu_{k+1}(x)\overset{(ii)}{=}\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{x}_{k+1})\overset{(*)}{=}\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{x}_k)=$
    $(\sigma(\mathtt{z}_k)-\sigma(\mathtt{x}_k))+(\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{z}_k))\overset{(IH)}{=}\nu_k(x)+t_{k+1}$
- $\nu_k+t_{k+1}\models cc$: for $cc=x\sim c$,[8] $\mathtt{cc}_{k+1\Delta}=(\mathtt{z}_{k+1}-\mathtt{x}_k)\sim c$. Because $\sigma\models\mathtt{cc}_{k+1\Delta}$ (see $(*)$), in particular $(\sigma(\mathtt{z}_k)-\sigma(\mathtt{x}_k))\sim c$ $(\dagger)$, and thus $\nu_k(x)+t_{k+1}\overset{\text{IH},(v)}{=}(\sigma(\mathtt{z}_k)-\sigma(\mathtt{x}_k))+(\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{z}_k))=\sigma(\mathtt{z}_k)-\sigma(\mathtt{x}_k)\overset{(\dagger)}{\sim}c$.
- $P_{k+1}\overset{(iii)}{=}\bigcup_{\sigma(\mathtt{A}_{k+1})=\mathtt{tt}}\mathtt{A}_{k+1}\overset{(*)}{=}P$
- $\delta_{k+1}\models dc$: for $dc=(d_A=d_i)$, we have $\delta_{k+1}(A)\overset{(iv),(*)}{=}\Delta^{-1}(\mathtt{n}^{\mathtt{i}})\overset{\text{def},\Delta}{=}\Delta^{-1}(\Delta(d_i))=d_i$, thus $\delta_{k+1}\models(d_A=d_i)$. For $dc=(d_A=d_B)$, we have $\mathtt{dc}_{k+1}\overset{\text{def},\mathtt{dc}}{=}(\mathtt{DA}_{k+1}=\mathtt{DB}_{k+1})$. Since $\sigma\models\mathtt{dc}_{k+1}$ (because of $(*)$), in particular $\sigma(\mathtt{DA}_{k+1})\overset{(*)}{=}\sigma(\mathtt{DB}_{k+1})\overset{(iv)}{=}\mathtt{n}^{\mathtt{i}}$ for some $\mathtt{n}^{\mathtt{i}}$, thus $\delta_{k+1}(A)\overset{(iv),(*)}{=}\Delta^{-1}(\mathtt{n}^{\mathtt{i}})\overset{(iv),(*)}{=}\delta_{k+1}(B)$, and therefore $\delta_{k+1}\models(d_A=d_B)$.

Thus, $\langle s,\nu_k\rangle\xrightarrow{P,\delta_{k+1},t_{k+1}}\langle s',\nu_k+t_{k+1}[X:=0]\rangle$ is obtained from e using (5).

Case $\sigma\models\varphi^{invisible}(e)_{k+1/t}$ $(**)$: let $e=(s,\emptyset,\mathtt{true},cc,X,s')$ (cf. (10)), then

- $l_k\overset{\text{IH}}{=}s$, $l_{k+1}\overset{(i)}{=}s'_{k+1}$, $t_{k+1}\overset{(v)}{=}\sigma(\mathtt{z}_{k+1})-\sigma(\mathtt{z}_k)$, ν_{k+1}, and $\nu_k+t_{k+1}\models cc$ as before
- $P_{k+1}\overset{(iii)}{=}\bigcup_{\sigma(\mathtt{A}_{k+1})=\mathtt{tt}}\mathtt{A}_{k+1}\overset{(**)}{=}\emptyset$
- $\delta_{k+1}\overset{(iv)}{=}\emptyset$, because $\sigma(\mathtt{A}_{k+1})\overset{(**)}{=}\mathtt{ff}$ and $\sigma(\mathtt{DA}_{k+1})\overset{(**)}{=}\mathtt{n}^{\perp}$ for all A
- $\delta\models\mathtt{true}$ for all δ.

Thus, $\langle s,\nu_k\rangle\xrightarrow{\emptyset,\emptyset,t_{k+1}}\langle s',\nu_k+t_{k+1}[X:=0]\rangle$ is obtained from e using (5) in case $t_{k+1}>0$, and using (6) in case $t_{k+1}=0$.

---

6 For each k, there exists exactly one such location s, cf. (12).

7 Δ is injective, with $\Delta(d_i)=\mathtt{n}^{\mathtt{i}}$ (cf. Section 3), thus $\mathtt{n}^{\mathtt{i}}\in range(\Delta)$.

8 Here and in the remainder of the proof, we only show the basic cases for clock and data constraints, but the results directly carry over to Boolean combinations of these.

Finally, we get $r_\sigma \in Run_{\mathcal{T},k+1}$, and we define $\tau : \mathcal{V}(\varphi(\mathcal{T})_k) \to Run_{\mathcal{T},k}$ such that for every interpretation $\sigma \in \mathcal{V}(\varphi(\mathcal{T})_k)$, $\tau(\sigma) = r_\sigma \in Run_{\mathcal{T},k}$ is the derived run.

Proposition 4.4 (*Derived Run, Product*). *For $\sigma \in \mathcal{V}(\varphi(\mathcal{T}_1 \bowtie \mathcal{T}_2)_k)$, the derived run r_σ is a run of $\mathfrak{S}_{\mathcal{T}_1 \bowtie \mathcal{T}_2}$ of length k, i.e., $r_\sigma \in Run_{\mathcal{T}_1 \bowtie \mathcal{T}_2, k}$.*

Proof (*Sketch*). The proof is along the same lines as the proof of Lemma 4.3. In **IS**, we first show that for $i = 1, 2$, reducing the transition

$$\langle (l_{k,1}, l_{k,2}), \nu_1 \rangle \xrightarrow{P_{k+1}, \delta_{k+1}, t_{k+1}} \langle (l_{k+1,1}, l_{k+1,2}), \nu_{k+1} \rangle,$$

to the constituents of $\mathcal{T}_i$ yields a transition $e_i = (l_{k,i}, P_{k+1,i}, dc_i, cc_i, X_i, l_{k+1,i})$ of $\mathcal{T}_i$, with $P_{k+1,i} \subseteq P_{k+1}$ (remember that delay transitions are a special case of invisible transitions). We then argue that all possible combinations of e_1 and e_2 correspond to a valid execution in the product automaton (Definition 2.5):

- Two visible transitions corresponds to parallel execution, that means to a transition of the form (1).
- A delay transition together with an (in)visible transition corresponds to one automaton performing a local (in)visible transition, while the other remains in its current location, thus, to a transition of the form (1).
- Two delay transitions correspond to a delay step of the product automaton, which – though not explicitly specified in Definition 2.5 – can be combined with the following transition to yield a transition of either form (1) or (2).
- A visible and an invisible transition correspond to a set of sequences of transitions of the form (2), each sequence comprising first the visible transition with delay d, followed by the invisible transition with delay d', $d > 0$, $d' \geq 0$. Similarly for two invisible transitions, with delays d and d', $d, d' \geq 0$.

Finally, we get $r_\sigma \in Run_{\mathcal{T}_1 \bowtie \mathcal{T}_2, k}$.

We now show that the formula representation is complete, i.e., for every run $r \in Run_{\mathcal{T},k}$, we can find a model $\sigma \in \mathcal{V}(\varphi(\mathcal{T})_k)$.

Definition 4.5 (*Derived Interpretation*). For $r \in Run_{\mathcal{T},k}$, the *derived interpretation σ_r over (the variables in) $\varphi(\mathcal{T})_k$* is

$$\sigma_r(\mathtt{s}_{k'}) = \mathtt{tt} \text{ iff } s = l_{k'}, \mathtt{ff} \text{ otherwise} \tag{vi}$$

$$\sigma_r(\mathtt{z}_{k'}) = 0 \text{ iff } k' = 0, \sigma_r(\mathtt{z}_{k'-1}) + t_{k'} \text{ otherwise} \tag{vii}$$

$$\sigma_r(\mathtt{x}_{k'}) = \sigma_r(\mathtt{z}_{k'}) - \nu_{k'}(x) \tag{viii}$$

$$\sigma_r(\mathtt{A}_{k'}) = \mathtt{tt} \text{ iff } k' > 0 \text{ and } A \in P_{k'}, \mathtt{ff} \text{ otherwise} \tag{ix}$$

$$\sigma_r(\mathtt{DA}_{k'}) = \Delta(\delta_{k'}(A)) \text{ iff } k' > 0 \text{ and } A \in P_{k'}, \mathtt{n}^\perp \text{ otherwise} \tag{x}$$

for all $0 \leq k' \leq k$. The derived interpretation for products of TCA is defined in the same way, except for rewriting (vi) to

$$\sigma_r(\mathtt{s}_{k'}) = \mathtt{tt} \text{ iff } s = l_{k',i}, s \in S_i, i = 1, 2, \text{ and } \mathtt{ff} \text{ iff } s \neq l_{k',i}, s \in S_i, i = 1, 2 \tag{vi'}$$

Lemma 4.6 (*Completeness*). *For $r \in Run_{\mathcal{T},k}$, the derived interpretation σ_r is a model of the k-unfolding of $\mathcal{T}$, that means $\sigma_r \models \varphi(\mathcal{T})_k$.*

Proof. Induction on k.

IA $r = \langle l_0, \nu_0 \rangle$: $\sigma_r(\bar{\mathtt{s}}_0) \overset{(vi)}{=} \mathtt{tt}$ for $l_0 = \bar{s}$ (initial location $\bar{s}$), $\sigma_r(\mathtt{s}_0) \overset{(vi)}{=} \mathtt{ff}$ otherwise. $\sigma_r(\mathtt{z}_0) \overset{(vii)}{=} 0$, $\sigma_r(\mathtt{x}_0) \overset{(viii)}{=} \sigma_r(\mathtt{z}_0) - \nu_0(x) = 0$ for all $x \in \mathcal{X}$, and $\sigma_r(\mathtt{A}_0) \overset{(ix)}{=} \mathtt{ff}$ and $\sigma_r(\mathtt{DA}_0) \overset{(x)}{=} \mathtt{n}^\perp$ for all $A \in \mathcal{P}$. For $I(\bar{s}) = x \sim c$,[9] $\mathtt{I}(\bar{\mathtt{s}})_0 = \mathtt{z}_0 - \mathtt{x}_0 \sim c$. Because $\nu_0 \models I(\bar{s})$ (Definition 2.8), in particular $0 = \nu_0(x) \sim c$ (‡). So, $\sigma_r(\mathtt{x}_0) \overset{‡}{\sim} c$, that means $\sigma_r \models \mathtt{I}(\bar{\mathtt{s}})_0$, thus $\sigma_r \models \varphi^{init}(\mathcal{T})$, and so $(\varphi(T)_0 = \varphi^{init}(\mathcal{T})) \, \sigma_r \models \varphi(T)_0$.

IH $r = \langle l_0, \nu_0 \rangle \xrightarrow{P_1, \delta_1, t_1} \cdots \xrightarrow{P_k, \delta_k, t_k} \langle l_k, \nu_k \rangle$: $\sigma_r \models \varphi(\mathcal{T})_k$, for $k \geq 0$.

IS $r = \langle l_0, \nu_0 \rangle \xrightarrow{P_1, \delta_1, t_1} \cdots \xrightarrow{P_k, \delta_k, t_k} \langle l_k, \nu_k \rangle \xrightarrow{P_{k+1}, \delta_{k+1}, t_{k+1}} \langle l_{k+1}, \nu_{k+1} \rangle \in Run_{\mathcal{T},k+1}$: let e be the transition of $\mathcal{T}$ underlying $\langle l_k, \nu_k \rangle \xrightarrow{P_{k+1}, \delta_{k+1}, t_{k+1}} \langle l_{k+1}, \nu_{k+1} \rangle$ (visible or invisible), $e = (s, P, dc, cc, X, s')$.

Case visible: $P_{k+1} \neq \emptyset$, $\delta_{k+1} \neq \emptyset$, and $t_{k+1} > 0$. Then

- $\sigma_r(\mathtt{s}_{k+1}) \overset{(vi)}{=} \mathtt{tt}$ iff $s = l_{k+1}$, and $\mathtt{ff}$ otherwise.
- $\sigma_r(\mathtt{z}_{k+1}) \overset{(vii)}{=} \sigma_r(\mathtt{z}_k) + t_{k+1} > \sigma_r(\mathtt{z}_k)$
- $\sigma_r(\mathtt{x}_{k+1}) \overset{(viii)}{=} \sigma_r(\mathtt{z}_{k+1}) - \nu_{k+1}(x) =$
$$\begin{cases} \overset{x \in X}{=} \sigma_r(\mathtt{z}_{k+1}) - 0 = \sigma_r(\mathtt{z}_{k+1}) \\ \overset{x \notin X, (vii)}{=} \sigma_r(\mathtt{z}_k) + t_{k+1} - \nu_{k+1}(x) \overset{\text{def. } \nu}{=} \\ \qquad \sigma_r(\mathtt{z}_k) + t_{k+1} - \nu_k(x) - t_{k+1} = \\ \qquad \sigma_r(\mathtt{z}_k) - \nu_k(x) \overset{IH}{=} \sigma_r(\mathtt{x}_k) \end{cases}$$
- $\sigma_r(\mathtt{A}_{k+1}) \overset{(ix)}{=} \mathtt{tt}$ iff $A \in P_{k+1}$, and $\mathtt{ff}$ otherwise

[9] Here and in the remainder of the proof, again we only show the basic cases for data and clock constraints.

- $\sigma_r(\mathtt{DA}_{k+1}) \overset{(x)}{=} \Delta(\delta_{k+1}(A))$ iff $A \in P_{k+1}$, and $\mathtt{n}^{\perp}$ otherwise
- For $dc = (d_A = d_i)$, we have $\mathtt{dc}_{k+1} = \mathtt{A}_{k+1} \wedge (\mathtt{DA}_{k+1} = \mathtt{n}^{i})$. Because $\delta_{k+1} \models dc$ (Definition 2.8), especially $\delta_{k+1}(A) = d_i$, i.e. $\sigma_r(\mathtt{DA}_{k+1}) \overset{(x)}{=} \Delta(\delta_{k+1}(A)) = \Delta(d_i) = \mathtt{n}^{i}$, and therefore $\sigma_r \models \mathtt{dc}_{k+1}$.
  For $dc = (d_A = d_B)$, we have $\mathtt{dc}_{k+1} \overset{\text{def.dc}}{=} \mathtt{A}_{k+1} \wedge \mathtt{B}_{k+1} \wedge (\mathtt{DA}_{k+1} = \mathtt{DB}_{k+1})$. Because $\delta_{k+1} \models dc$ (Definition 2.8), we have $\delta_{k+1}(A) = \delta_{k+1}(B) = d_i$, for some $d_i \in \mathcal{D}ata$. Thus $\sigma_r(\mathtt{DA}_{k+1}) \overset{(x)}{=} \Delta(\delta_{k+1}(A)) = \Delta(d_i) = \mathtt{n}^{i}$, and also $\sigma_r(\mathtt{DB}_{k+1}) \overset{(x)}{=} \Delta(\delta_{k+1}(B)) = \Delta(d_i) = \mathtt{n}^{i}$, and therefore $\sigma_r \models \mathtt{dc}_{k+1}$.
- For $I(s) = (x \sim c)$, we have $\mathtt{I(s)}_{k+1\Delta} = (\mathtt{z}_{k+1} - \mathtt{x}_{k}) \sim c$. Because $\nu_k + t \models I(s)$ for all $0 \le t \le t_{k+1}$ (Definition 2.8), in particular $\nu_k(x) + t_{k+1} \models (x \sim c)$; thus $\sigma_r(\mathtt{z}_{k+1}) - \sigma_r(\mathtt{x}_{k}) \overset{\text{IH},(viii)}{=} (\sigma_r(\mathtt{z}_{k}) + t_{k+1}) - (\sigma_r(\mathtt{z}_{k}) + \nu_k(x) = \nu_k(x) + t_{k+1} \sim c$, and therefore $\sigma_r \models \mathtt{I(s)}_{k+1\Delta}$. The argumentation for $\sigma_r \models \mathtt{cc}_{k+1\Delta}$ and $\sigma_r \models \mathtt{I(s')}_{k+1}$ is similar.

From the above, we get $\sigma_r \models \varphi^{visible}(e)_{k+1/t}$ (9) (so $\sigma_r \models \varphi^{trans}(\mathcal{T})_{k+1/t}$ (11)), $\sigma_r \models \varphi^{location}(\mathcal{T})_{k+1/t}$ (12), and $\sigma_r \models \varphi^{data\_value}(\mathcal{T})_{k+1/t}$ (13).

Case invisible: $P_{k+1} = \emptyset$, $\delta_{k+1} = \emptyset$, $dc = \mathtt{true}$, and $t \ge 0$. Then

- $\sigma_r(\mathtt{s}_{k+1})$ and $\sigma_r(\mathtt{x}_{k+1})$ as before.
- $\sigma_r(\mathtt{z}_{k+1}) \overset{(vii)}{=} \sigma_r(\mathtt{z}_{k}) + t_{k+1} \ge \sigma_r(\mathtt{z}_{k})$
- $\sigma_r(\mathtt{A}_{k+1}) \overset{(ix)}{=} \mathtt{ff}$, and $\sigma_r(\mathtt{DA}_{k+1}) \overset{(x)}{=} \mathtt{n}^{\perp}$ for all A.
- $\sigma_r \models \mathtt{I(s)}_{k+1\Delta}$, $\sigma_r \models \mathtt{cc}_{k+1\Delta}$, and $\sigma_r \models \mathtt{I(s')}_{k+1}$ as before.
- The data constraint $\mathtt{true}$ is trivially satisfied.

From the above, we get $\sigma_r \models \varphi^{invisible}(e)_{k+1/t}$ (10) (so $\sigma_r \models \varphi^{trans}(\mathcal{T})_{k+1/t}$ (11)), $\sigma_r \models \varphi^{location}(\mathcal{T})_{k+1/t}$ (12), and $\sigma_r \models \varphi^{data\_value}(\mathcal{T})_{k+1/t}$ (13).

Finally, we get $\sigma_r \models \varphi(T)_{k+1}$, and we define $\varsigma : Run_{\mathcal{T},k} \to \mathcal{V}(\varphi(\mathcal{T})_k)$ such that for every run $r \in Run_{\mathcal{T},k}$, $\varsigma(r) = \sigma_r \in \mathcal{V}(\varphi(\mathcal{T})_k)$ is the derived interpretation.

Proposition 4.7 (*Derived Interpretation, Product*). *For $r \in Run_{\mathcal{T}_1 \bowtie \lhd \mathcal{T}_2,k}$, the derived interpretation σ_r is a model of $\varphi(\mathcal{T}_1 \bowtie \lhd \mathcal{T}_2)_k$, i.e. $\sigma \in \mathcal{V}(\varphi(\mathcal{T}_1 \bowtie \lhd \mathcal{T}_2)_k)$.*

Proof (*Sketch*). The proof is along the same lines as the proof of Lemma 4.6. In **IS**, we show that for $i = 1, 2$, the derived interpretation σ_r for $r \in Run_{\mathcal{T}_1 \bowtie \lhd \mathcal{T}_2,k+1}$, reduced to the variables of $\varphi(\mathcal{T}_i)_k$, is a model of $\varphi(\mathcal{T}_i)_k$. Because delay transitions are special cases of invisible transitions (cf. Section 3.4), we directly get $\sigma \in \mathcal{V}(\varphi(\mathcal{T}_1 \bowtie \lhd \mathcal{T}_2)_k)$.

Using the above, we can now show the main theorem.

Theorem 4.8 (*Soundness, Completeness*). *The representation TCA, as defined in Definition 3.1, is sound and complete.*

Proof. This follows directly from Lemmas 4.3 and 4.6.

5. Abstraction

Having defined a framework for exhaustive BMC of TCA in Section 3, we now show how to further increase the manageable system size, and overcome the state explosion problem, by using abstraction refinement techniques. We show how to adapt the abstraction technique of *abstraction by merging omission (MO)* [22] to our representation. MO is a simple and fast but nevertheless powerful abstraction technique specifically tailored to work on logical formulae. The removal of constraints considered irrelevant to the particular safety property yields an over-approximation.

5.1. Abstraction by merging omission

The basic idea of MO is to reduce the system complexity by decreasing the number of symbols in $\varphi(\mathcal{T})$, while retaining as much information about the TCA transition characteristics as possible (the abstract formula is weaker than $\varphi(\mathcal{T})$, though). It is defined for formulae in negation normal form (NNF), to which $\varphi(\mathcal{T})$ can be easily transformed. MO uniformly works on the different syntactical categories: it merges (Boolean) location and (natural) port variables, by mapping them to the same image according to a *map of merging*, and it removes (rational) clock variables and arithmetic constraints according to a *set of omission*.

Definition 5.1 (*Abstraction by Merging Omission*). Let $\mathcal{T}$ be a TCA, let $\varphi(\mathcal{T})$ be in NNF. Let $\mathtt{S}$, $\mathtt{X}$, $\mathtt{P}_{\mathtt{A}}$ and $\mathtt{P}_{\mathtt{DA}}$ be the variable sets representing locations, clocks, port activity variables and port data variables, respectively, all without indices; let $\mathtt{P} = (\mathtt{S} \cup \mathtt{P}_{\mathtt{A}})$, let $\vartheta : \mathtt{P}_{\mathtt{DA}} \to \mathtt{P}_{\mathtt{A}}$ a mapping such that $v \in \mathtt{P}_{\mathtt{DA}}$ and $\vartheta(v) = v' \in \mathtt{P}_{\mathtt{A}}$ are data and activity variable of the same port. Let $\mathcal{O} \subseteq \mathtt{X} \cup CC(\mathtt{X}) \cup DC(\mathtt{P}_{\mathtt{DA}})$ be a set not containing compound formulae, let $\gamma : \mathtt{P} \to \mathtt{P} \dot{\cup} \mathtt{P}'$ be a mapping, with $\mathtt{P}'$ some fresh set of propositional variables, and $\gamma(p) = \gamma(p')$ only if $(p, p' \in \mathtt{S})$ or $(p, p' \in \mathtt{P}_{\mathtt{A}})$ (i.e., γ cannot merge a location with a port).

The *abstraction by merging omission with respect to $\mathcal{O}$ and γ* (or simply *abstraction*) of $\varphi(\mathcal{T})$, denoted as $\alpha_{\mathcal{O},\gamma}(\varphi(\mathcal{T}))$, is defined in (26). We may omit $\mathcal{O}$ and γ in $\alpha_{\mathcal{O},\gamma}(\varphi(\mathcal{T}))$ if they are clear form the context.

$$\alpha'(L) = \begin{cases} L & L{=}\neg p,\ p{\in}\mathrm{P},\ \gamma(p){=}id & (23a)\\ L & cont(L)\cap(\mathcal{O}\cup\mathrm{P}) = \emptyset,\ \forall v{\in}(cont(L){\cap}\mathrm{P}_{\mathrm{DA}}){:}\gamma(\vartheta(v)){=}id & (23b)\\ \gamma(L) & L{=}p{\in}\mathrm{P} & (23c)\\ \gamma(L) & L{=}\neg p,\ p{\in}\mathrm{P},\ \exists p'{\in}\mathrm{P} \text{ in the same conjunct as } p : \gamma(p){=}\gamma(p') & (23d)\\ \neg\gamma(L) & L{=}\neg p,\ p{\in}\mathrm{P},\ \nexists p'{\in}\mathrm{P} \text{ in the same conjunct as } p : \gamma(p){=}\gamma(p'),\\ & \exists \neg p'', p''{\in}\mathrm{P},\ \text{in the same conjunct as } p : \gamma(p){=}\gamma(p'') & (23e)\\ \texttt{true} & \text{otherwise} & (23f) \end{cases}$$

$$\alpha'(F \wedge G) = \alpha'(F) \wedge \alpha'(G) \quad (24a)$$

$$\alpha'(F \vee G) = \alpha'(F) \vee \alpha'(G) \quad (24b)$$

$$\gamma_\alpha = \bigwedge_{p\in\gamma(\mathrm{P})} \Big(\big(\bigvee_{p'\in\gamma^{-1}(p)} p' \big) \leftrightarrow p \Big) \quad (25)$$

$$\alpha(\varphi(\mathcal{T})) = \alpha'(\varphi(\mathcal{T})) \wedge \gamma_\alpha \quad (26)$$

Here, F and G are formulae in NNF, L a literal, $cont(L)$ the set of atomic formulae and variables occurring in L.

Fig. 6. Abstraction by merging omission.

MO uniformly captures abstraction on all syntactic categories contained in $\varphi(\mathcal{T})$: negative propositional variables not meant to be abstracted (i.e., where γ is the identity) are kept unchanged (23a), so are clock and data constraints (positive or negative) not contained in $\mathcal{O}$ (23b). However, we may retain only those data constraints that reason about ports not merged by γ, since the conjunction of data constraints may become unsatisfiable otherwise.[10] This is ensured by the constraint $\forall v{\in}(cont(L){\cap}\mathrm{P}_{\mathrm{DA}}){:}\gamma(\vartheta(v)){=}id$ in (23b).

The map γ is applied to all positive propositional variables (23c). For negative propositional variables p (i.e., which occur as $\neg p$) meant to be abstracted, we distinguish two cases: if there exists a positive propositional variable p' in the same innermost conjunct/disjunct as p, and with the same image under γ (i.e., p and p' are to be merged), we replace p with its positive image under γ (23d). The idea is that positive propositional variables are used to describe the "behaviour" – source and target of a transition, for example – while negative propositional variables are used to ensure consistency – mutual location exclusion, for example. Therefore, if such p' exists, we can dismiss the literal $\neg p$, since p and p' are mapped to the same image under γ, and we do not need the consistency constraint anymore. Note that replacing $\neg p$ by `true` is possible as well, but this would yield a much coarser abstraction. If no such p' exists, but a instead a negative propositional variable p'' with the same image under γ (23e), then we replace p and p'' by their negative image under γ. In all other cases, α maps the literal to `true` (23f). In this way, α performs a quick variant of existential abstraction [14], while exploiting the structural relationships of clocks and TCA, and taking advantage of our formula representation.

In order to guarantee that MO yields an over-approximation, we further need to keep track of the relation between symbols in P and their abstract counterparts in P′. For this reason, we add the constraint γ_α (25) to the abstract formula.

Remark 5.2 (*Notation*)**.** Without confusion, in what follows we use the symbol α only, and omit the symbol α'. For example, for a literal L, $\alpha(L)$ denotes $\alpha'(L)$.

With this, we get the following lemma.

Lemma 5.3 (*Abstraction by Weakening*)**.** *MO yields an over-approximation, that means $\alpha(F)$ is weaker than F in the sense that the implication $F{\rightarrow}\alpha(F)$ is* valid *(true in all models).*

Lifting α to the presence of localisations is straightforward: γ and $\mathcal{O}$ are understood oblivious to indices in the NNF of $\varphi(\mathcal{T})$, such that indices directly carry over to $\varphi(\mathcal{T})_k$ unchanged (defining different abstractions for different steps is possible using the same definition of α but we consider it to be less useful). Note that α is homomorphic with respect to $\{\wedge, \vee\}$, which proves the equality of $\alpha(\varphi(\mathcal{T})_k)$ and $\alpha(\varphi(\mathcal{T}))_k$ (except for speed of computing the abstraction, where $\alpha(\varphi(\mathcal{T}))_k$ is superior).

The major difference between our abstraction function and the one presented in [22] is the fact that we do *not* in general map negative propositional variables to `true`, since this would effectively remove the mutual exclusion constraint for locations (12) as well as part of the consistency constraint on data values (13). Instead, we take into account the special characteristics of our formula representation, and the syntactic context of the propositional variable (23d), (23e), and in this way retain more information. See Section 5.2 for further details.

[10] Suppose a transition f with port set $\{A, B\}$ and data constraint $((d_A{=}1){\wedge}(d_B{=}2))$, and ports A and B are to be merged into port C. Straightforward syntactic replacement of port data variables would yield a transition f' with port set $\{C\}$ and data constraint $((d_C{=}1){\wedge}(d_C{=}2))$. While the data constraint of f is satisfiable, the data constraint of f' is not satisfiable; such abstraction would not yield an over-approximation anymore.

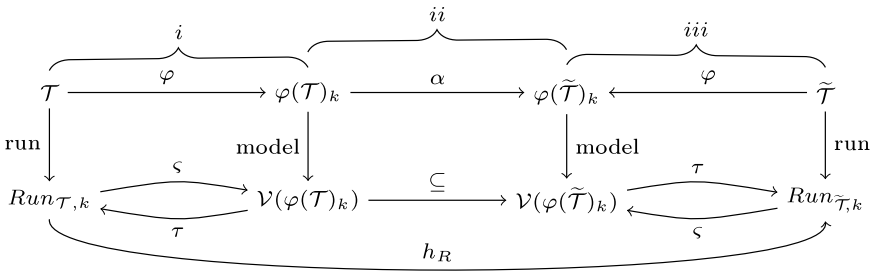

Fig. 7. Strong correctness of abstraction.

Example 5.4 (*Abstraction*)**.** Consider again the formula representation of component $\mathcal{C}$, presented in Example 3.3. To abstract from timing information, we choose $\mathcal{O}=\{y\}$, and $\gamma=id$. The result of applying α with respect to γ and $\mathcal{O}$ to the formula representation $\varphi(\mathcal{C})$ (15) is shown in (27).

$$\begin{aligned}
\alpha(\varphi^{init}(\mathcal{C})) &= \mathtt{c}_0 \wedge \neg \mathtt{A}_0 \wedge (\mathtt{DA}_0{=}0) \wedge (\mathtt{z}_0{=}0)\\
\alpha(\varphi^{visible}(\mathcal{C})) &= (\mathtt{c}_{t\text{-}1} \wedge \mathtt{A}_t \wedge (\mathtt{DA}_t{=}1) \wedge (\mathtt{z}_{t\text{-}1}{<}\mathtt{z}_t) \wedge\\
&\qquad \mathtt{c}_t) \vee (\mathtt{c}_{t\text{-}1} \wedge \mathtt{A}_t \wedge (\mathtt{DA}_t{=}2) \wedge (\mathtt{z}_{t\text{-}1}{<}\mathtt{z}_t) \wedge \mathtt{c}_t)\\
\alpha(\varphi^{trans}(\mathcal{C})) &= \alpha(\varphi^{visible}(\mathcal{C}))\\
\alpha(\varphi^{data_value}(\mathcal{C})) &= (\mathtt{DA}_t{\leq}2) \wedge (\neg \mathtt{A}_t \vee \neg(\mathtt{DA}_t{=}0)) \wedge (\mathtt{A}_t \vee (\mathtt{DA}_t{=}0))\\
\alpha(\varphi(C)) &= \alpha(\varphi^{init}(\mathcal{C})) \wedge \alpha(\varphi^{trans}(\mathcal{C})) \wedge \alpha(\varphi^{data_value}(\mathcal{C}))
\end{aligned} \tag{27}$$

5.2. Correctness

For α to yield a *correct over-approximation*, every finite run of the concrete system $\mathcal{T}$ (represented by a model of $\varphi(\mathcal{T})_k$, see Theorem 4.8) has to be reproducible in the abstract case, which is already captured in Lemma 5.3. Here, we prove an even stronger correctness result, which in particular emphasises the structural relationships between concrete and abstract formula: By showing that Fig. 7 commutes, we conclude the existence of a homomorphism h_R between concrete and abstract set of runs.

The idea of the proof is as follows: since α works locally, it retains the formula structure of (14). In particular, the structure of clock and data constraints is preserved, as these constraints either remain in the formula unchanged or are replaced by `true`. Therefore, there exists some TCA $\widetilde{\mathcal{T}}$ of the same representation $\varphi(\widetilde{T})_k = \alpha(\varphi(T)_k)$ (up to logical equivalence), and the subdiagrams (i) and (iii) commute according to Theorem 4.8. According to Lemma 5.3, the subdiagram (ii) commutes as well (as every model of $\varphi(\widetilde{T})_k$ is a model of $\alpha(\varphi(T)_k)$), such that the whole diagram commutes. Hence, the existence of a homomorphism h_R is a direct consequence, and we get

Theorem 5.5 (*Correctness of Abstraction*)**.** *MO, as defined in Definition 5.1, yields a correct over-approximation on sets of runs.*

For a detailed discussion and proof (including the proof of Lemma 5.3), we refer to the extended version of this paper, available at www.cwi.nl/~kemper.

5.3. Abstraction refinement

In this section, we give a brief overview of our abstraction refinement methodology. The general abstraction refinement paradigm [14] consists of three steps: (1) generate the initial abstraction, (2) model check the abstract system, and, if required, (3) refine the abstraction.

Generate the initial abstraction. If there is no additional knowledge about the system, the initial abstraction simply removes from $\varphi(\mathcal{T})$ all symbols in $CC(\mathtt{X}) \cup DC(\mathtt{P}_{\mathtt{DA}})$, and merges all symbols in $\mathtt{S}$ to a single one (we refer to [14] for improved techniques), thereby collapsing to a single trivial location (accordingly for ports). Yet, the next refinement iterations will quickly discover more relevant parameters.

Model checking the abstract system. If $\alpha(\varphi(\mathcal{T}))_k$, together with a representation ρ of the safety property (cf. Section 3.3), is unsatisfiable, the system is safe within bound k (cf. Definition 2.9, Theorem 4.8 and Theorem 5.5). Otherwise, the counterexample needs to be *concretised*, which amounts to checking $\varphi(\mathcal{T})_k \wedge \rho$, in conjunction with the variable valuations π representing the abstract counterexample, and concretising constraints of the form $\mathtt{u} \rightarrow \mathtt{s} \vee \mathtt{r}$ for all locations and ports s and r with $\gamma(\mathtt{s})=\gamma(\mathtt{r})=\mathtt{u}$. This check can be done very quickly, since the single abstract counterexample is highly restrictive. If the conjunction is satisfiable, a counterexample to the property is found. Otherwise, the counterexample is spurious, and the abstraction needs to be refined.

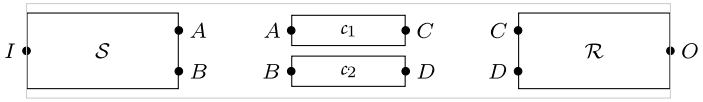

Fig. 8. ABP Connector, conceptual overview.

Refining the abstraction. To identify ill-abstracted parameters, we stratify the formulae $\varphi(\mathcal{T})_k$, ρ and π (i.e., align them along their unfolding depth k), and derive a sequence of Craig interpolants (e.g. [25]),[11] one for every bisection into prefix and suffix. By definition, both the prefix of the first interpolant $G \equiv \texttt{false}$ and the suffix of the last interpolant $\widetilde{G} \equiv \texttt{true}$ are unsatisfiable, and, for P being the set of symbols subject to abstraction, at least one of the symbols in $\mathbb{A} \stackrel{def}{=} cont(G) \cap P$ has been wrongly abstracted.

The difficulty – in particular in automatic abstraction refinement – is then to define heuristics describing the application of the two refinement strategies (a) refine a symbol from $\mathbb{A}$, and (b) rule out the (sub)run represented by the common parts of the prefix of G and the suffix of $\widetilde{G}$. The former quickly collapses to the concrete system if applied too frequently, while the latter cannot yield results as long as essential parameters are inadequately abstracted. Thus, it is necessary to define heuristics that strike a suitable balance between (a) and (b).

The *fully automatic* heuristic presented in [22] (together with its optimisations) is a compromise between the drawbacks of the two alternatives: after refining a parameter (a), a fixed number of runs (fractions of the unfolding depth k have turned out to be most promising) is ruled out (b) before refining the next symbol according to (a).

5.4. Discussion

We do not have to distinguish between abstraction of different constituents of TCA, since α works uniformly depending just on the different syntactical categories (propositional, natural, real variables), which happen to represent different concepts of TCA. Yet, in contrast to [22], our abstraction function does not remove negative propositional variables from the formula in case the map of merging γ is the identity for these, and moreover retains more information for negative propositional which are to be abstracted. This speeds up the verification process, since we preserve a bigger part of the formula structure of $\varphi(\mathcal{T})$, which not only provides more meaningful results, but therefore also results in less cycles in the abstraction refinement loop.

Proving a strong correctness result in Section 5.2 permits to conclude the existence of a corresponding effective abstraction technique on TCA which produces $\widetilde{\mathcal{T}}$. Yet, the formalisation of the direct construction will be much less uniform than what has been presented here.

As a second major result of the strong correctness proof,[12] we get that every abstraction satisfying Lemma 5.3 is already proven correct in our framework. The existence of the abstract TCA $\widetilde{\mathcal{T}}$, however, is not a general consequence, but a particular result of our strong correctness. This makes α a very powerful and universal technique, yet it remains efficient due to its purely syntactical definition.

6. Example: alternating bit protocol

In this section, we present an example in more detail, to provide a better understanding of the framework presented in this paper. For each modelling/analysis step, we point out the available tool support, and in Section 6.4, we present some preliminary experimental results. All files mentioned are available at http://www.cwi.nl/~kemper/ABPexample/.

We model a network protocol, which ensures successful transmission of data elements between sender and receiver over unreliable channels. This so-called *alternating bit protocol (ABP)* is one of the standard benchmarks in the context of component-based systems and process algebra, and has been discussed in detail for example in [26,17,23].

Essentially following the description in [26], we design a compositional connector $\mathcal{ABP}$ from four subconnectors: the sender $\mathcal{S}$, the receiver $\mathcal{R}$, and two unreliable channels c_1 and c_2 connecting the former two, see Fig. 8. The channels may loose, but not corrupt or duplicate, data at random. Yet, it is very easy to change this behaviour, simply by exchanging the channels c_1 and/or c_2. For an overview of how to model timed channels with different behaviour, see e.g. [4]. The intended behaviour of $\mathcal{ABP}$ is as follows: after accepting input (from the network), $\mathcal{S}$ starts a timer, and sends the message to $\mathcal{R}$ via c_1. $\mathcal{S}$ attaches a control bit b to the message, and as acknowledgement expects from $\mathcal{R}$ the corresponding control bit b through c_2. After that, $\mathcal{S}$ is ready to accept another input from the environment, which it sends to $\mathcal{R}$ with attached control bit $\neg b$. If the timer expires before $\mathcal{S}$ receives the acknowledgement b, or if it receives an acknowledgement $\neg b$ (which it ignores), it resets the timer and resends the message with bit b.

[11] A Craig interpolant for an inconsistent pair of formulae (A, B) is a formula C that is implied by *prefix* A, inconsistent with *suffix* B and contains only common symbols of A and B; it is thus an over-approximation of A and an under-approximation of $\neg B$.

[12] Mainly subdiagrams (i) and (ii) in Fig. 7.

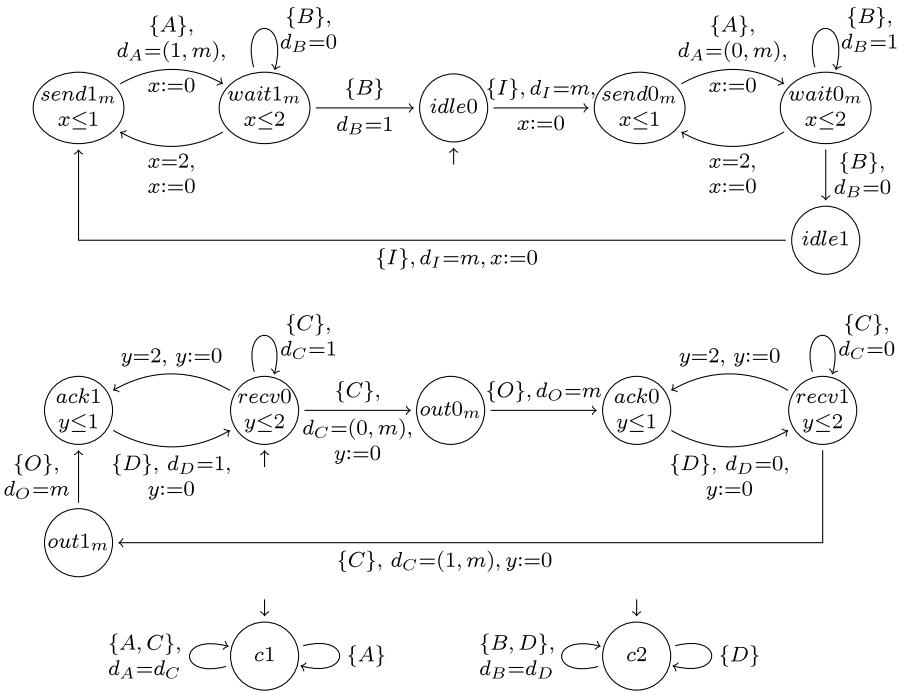

Fig. 9. Sender (top), receiver (middle), and connecting channels (bottom) of the ABP.

Component $\mathcal{R}$ works complementary: it receives a message, together with a control bit b, from $\mathcal{S}$. After delivering the message to the network, $\mathcal{R}$ sets a timer, and sends bit b as acknowledgement to $\mathcal{S}$. Next, it expects a message with bit $\neg b$. If the timer expires, or the next message is tagged with b again (which it ignores), $\mathcal{R}$ resets the timer and resends the acknowledgement b.

We assume an arbitrary but fixed, finite set of messages $\mathcal{M}sg$. The data domain is $\mathcal{D}ata = \mathcal{M}sg \cup \{0,1\} \times \mathcal{M}sg \cup \{0,1\}$. The elements of $\mathcal{D}ata$ correspond to messages sent from the environment to $\mathcal{S}$, and from $\mathcal{R}$ to the environment ($\mathcal{M}sg$), control bit/message pairs sent from $\mathcal{S}$ to $\mathcal{R}$ ($\{0,1\} \times \mathcal{M}sg$), and acknowledgements sent from $\mathcal{R}$ to $\mathcal{S}$ ($\{0,1\}$). The (data parametrised) TCA for $\mathcal{S}$, $\mathcal{R}$, c_1 and c_2 are shown in Fig. 9, where we use m to refer to any element of $\mathcal{M}sg$.[13] We have modelled the TCA in the editor available within the ECT [16] (file `ABP.ea`). For the data parametrisation, we used the "state memory extension".

6.1. Representation

The formula representation of $\mathcal{ABP}$ (cf. (20)) is given by

$$\varphi(\mathcal{ABP}) = \varphi(\mathcal{S} \bowtie \mathcal{R} \bowtie c_1 \bowtie c_2) = \varphi(\mathcal{S}) \wedge \varphi(\mathcal{R}) \wedge \varphi(c_1) \wedge \varphi(c_2).$$

From within the TCA editor, we can automatically generate the formula representation in MathSAT format. The editor offers customisation for a variety of parameters, including for example the unfolding depth, the size of the data domain, or selecting the automata for which to generate the formulae (one up to all). Moreover, the editor takes care of whether or not delay transitions need to be generated (only for two or more automata, cf. Section 3.4), and it offers to choose up to one location per automaton to generate a simple reachability property. In the future, a direct link to the MathSAT tool as well as an option to specify a more complex (reachability) property will be available.

For a system of this size, a comparably small unfolding depth of e.g. 20 is usually enough to check for reachability of error states. For performance comparison, and to show that our approach scales very well on reachability properties, we compare three unfolding depths, for $\mathtt{k} \in \{20, 50, 100\}$. Formula `ABP` in files `ABPk.msat` contains the representation of the k-unfolding $\varphi(\mathcal{ABP})_{\mathtt{k}}$. The formula representation is not really human readable (and actually is not intended to be, since it will become a mere internal format in the future). Yet, for the interested reader, we have added some comments to the files, to help understanding the representation.

[13] For example, $send0_m$ represents a set of $|\mathcal{M}sg|$ locations, one for each element of $\mathcal{M}sg$.

	k=20		k=50		k=100	
	¬Buffer	Error	¬Buffer	Error	¬Buffer	Error
ABPk	1.2s, 21.7MB	1.1s, 21.8MB	49.3s, 81.2MB	12.4s, 44.5MB	2976.5s, 815.2MB	77.8s, 108.5MB
ABPkabs1	1.4s, 21.86MB	0.7s, 20.4MB	887.6s, 301.3MB	5.9s, 35.8MB	segm. fault	28.2s, 66.4MB
ABPkabs2	1.3s, 19.6MB	-	208.7s, 189.7MB	-	segm. fault	-

All experiments have been carried out with MathSAT, on an Intel Core 2 Quad with 2.83GHz, 8GB RAM and Fedora 10.

Fig. 10. Preliminary experimental results.

6.2. BMC

While the internal behaviour of $\mathcal{ABP}$ ensures reliable transmission of messages over unreliable channels, from the outside, it behaves as a perfect buffer of capacity one (cf. [26]). That is, it accepts and delivers messages from and to the network alternately, and the order of data elements is not changed. The alternation is described by the LTL formula

$$\Box((I \rightarrow \bigcirc(\neg I \mathcal{U} O)) \wedge (O \rightarrow \bigcirc(\neg O \mathcal{U} I))), \tag{28}$$

which expresses that between any two communications through port I, there is a communication through port O, and vice versa.

Formula `Buffer` in files `ABPk.msat` contains the encoding of (28) for the corresponding depth, essentially following the approach of [6]. To reason about ultimately periodic runs, we assume a loop from step `k` back to step 1, encoded in formula `LOOPBACK`. We verify the above property by checking[14] the conjunction `ABP`∧¬`Buffer`∧`LOOPBACK`. The outcome is "unsatisfiable", which means that data flow through ports I and O alternates (literally: there exists *no run* in $\mathcal{ABP}$ where data flow does *not* alternate).

To check for the correct order of data elements, we identify a set of error states $\{(wait\mathtt{i}_m, out\mathtt{i}_n), (send\mathtt{i}_m, out\mathtt{i}_n)\}$, i=1, 2, $m, n \in \mathcal{M}sg$, $m \neq n$. Reachability of any of these states corresponds to a state of $\mathcal{ABP}$ where $\mathcal{S}$ has received data item m through I, but $\mathcal{R}$ is about to send data item n ($\neq m$) through O (we use the previous result that data flow through I and O alternates). Formula `Error` in files `ABPk.msat` contains the representation of the reachability property, cf. also Section 3.3. To check for reachability of any of the error states, we check the conjunction `ABP`∧`Error`. The result is "unsatisfiable", so none of the error states is reachable, and thus data items are sent to the environment in the same order as they were received.

6.3. Abstraction

The idea of abstraction is to increase the manageable system size and the speed of verification, by removing constraints which are considered irrelevant to the particular safety property (cf. Section 5). Timing information can be considered irrelevant for both `Buffer` and `Error`. Consequently, we define $\mathcal{O}_1 = \{x, y\}$ and $\gamma_1 = id$. The resulting formula representation `ABPabs1` of $\alpha_{\mathcal{O}_1,\gamma_1}(\varphi(\mathcal{ABP})_{\mathtt{k}})$ is given in files `ABPkabs1.msat`.

Since `Buffer` does not rely on exact data values, but only reasons about activity of ports, we may define an even coarser abstraction, which in addition removes information about the exact data values: $\mathcal{O}_2 = \{x, y, (d_I = m), (d_O = m)\}$, $\gamma_2(wait\mathtt{i}_m) = wait\mathtt{i}$, $\gamma_2(send\mathtt{i}_m) = send\mathtt{i}$, and $\gamma_2(out\mathtt{i}_m) = out\mathtt{i}$, for i=1, 2 and all $m \in \mathcal{M}sg$. The resulting formula representation `ABPabs2` of $\alpha_{\mathcal{O}_2,\gamma_2}(\varphi(\mathcal{ABP})_{\mathtt{k}})$ is given in files `ABPkabs2.msat`. Checking `ABPabs2`∧¬`Buffer`∧`LOOPBACK` (`Error` does not hold anymore) gives the expected result of "unsatisfiable" (as before).

The support for abstraction is not yet publicly available, though the theory is already implemented. A user interface for interactive abstraction (and refinement) will be available in future releases of the ECT.

6.4. Preliminary experimental results

Some preliminary experimental results are shown in Fig. 10.

The results clearly show that our approach is tailored to reachability properties, and that on these, it scales very well for large unfolding depths. While for k=20, the two properties take around the same time and memory consumption, checking `Error` is factor 4 faster, with factor 2 less memory, than `Buffer` for k=50, and almost factor 40 faster, with factor 8 less memory, for k=100. Comparing the same property on different unfolding depths, time and memory consumption increase by factors 40 and 4 (k=20 to k=50), and factors 60 and 10 (k=50 to k=100) for `Buffer`, while these factors are limited to 11 and 2 (k=20 to k=50) and 6 and 2.5 (k=50 to k=100) for `Error`.

[14] Using MathSAT, `mathsat -solve -input=msat -logic=QF_LRA -split_eq INPUTFILE`. See [24] for a detailed description of how to use the tool.

As a second result, Fig. 10 shows the improved performance for `Error` on the abstract system, while for `Buffer`, performance decreases, even leading to a segmentation fault for k=100. Though this might seem surprising at first glance, the reason is obvious: since `Buffer` reasons about all possible runs, and the abstract system permits more runs than the concrete system, checking `Buffer` on the abstract system is more expensive. What can be seen though is a slightly improved performance when comparing the two abstractions.

7. Conclusion and future work

In this paper, we have presented a SAT-based approach for bounded model checking of TCA. We have defined an embedding of bounded model checking for systems of TCA into propositional logic with linear arithmetic, and introduced a uniform logic-based abstraction for clocks, locations, port names and data values. This logical representation directly benefits from state-of-the-art SAT solving techniques, and allows a linear-size representation of parallel composition. We expect the structural relationships underlying the abstraction to provide the basis for a framework to generalise our work to other scenarios. For example, our approach can be straightforwardly applied to other automata models for coordination mechanisms, like for example intentional automata [15] or guarded automata [11].

Besides this, future work includes performance comparisons when using a logarithmic encoding for locations and ports (though automatic abstraction is more involved in that case), and the application and comparison of both variants on case studies. We want to improve the abstraction function, to retain even more information about mutual exclusion and data values, when abstracting from locations and ports. After having defined an *abstraction* that is tailored towards TCA in this paper, naturally the next step is to define tailor-made *refinement heuristics* for TCA, by exploiting the algebraical and logical principles underlying them. As a first step, we plan to add isomorphy inference reasoning to strategy (*b*) (cf. Section 5.3). We further plan to investigate how the notion of bisimulation carries over to the formula representation, and how it can be preserved under abstraction.

We believe our framework provides means to better understand the functioning of TCA, $\mathcal{R}$eo coordinators and $\mathcal{R}$eo networks [3] (for which TCA serve as formal model). To further improve this, we plan to integrate a back translation from formulae to TCA into the ECT (exploiting the fact that all transformations preserve the formula structure, cf. the correctness of abstraction proof for details), such that e.g. the result of abstraction can be viewed as a TCA in the editor. Our framework further facilitates verification of these connectors, for example whether an implementation meets its specification. We plan to implement a translation from LTL properties to our formula representation, using the encoding in [6]: the encoding is straightforward, but – unlike reachability properties – even for small properties it quickly reaches the limit of what can be done with pen and paper. We intend to use the framework within a testing environment of $\mathcal{R}$eo networks, which will enables us to perform black and white box testing (e.g., check the feasibility of a certain interaction behaviour), and support controller synthesis (bounding a subset of the variables, we get a constraint on solvability which includes non-hidden variables only).

Acknowledgement

Part of this research has been funded by the Dutch BSIK/BRICKS project.

References

[1] Rajeev Alur, Timed automata, in: N. Halbwachs, D. Peled (Eds.), CAV, in: LNCS, vol. 1633, Springer, 1999, pp. 8–22.
[2] Farhad Arbab, What do you mean, coordination? Bulletin of the Dutch Association for Theoretical Computer Science (NVTI) (1998) 11–22.
[3] Farhad Arbab, $\mathcal{R}$eo: a channel-based coordination model for component composition, Mathematical Structures in Computer Science 14 (3) (2004) 329–366.
[4] Farhad Arbab, Christel Baier, Frank S. de Boer, J.J.M.M. Rutten, Models and temporal logical specifications for timed component connectors, Software and System Modeling 6 (1) (2007) 59–82.
[5] Farhad Arbab, Christel Baier, J.J.M.M. Rutten, M. Sirjani, Modeling component connectors in $\mathcal{R}$eo by constraint automata (extended abstract), Electronic Notes in Theoretical Computer Science 97 (2004) 25–46.
[6] G. Audemard, A. Cimatti, A. Kornilowicz, R. Sebastiani, Bounded model checking for timed systems, in: D. Peled, M.Y. Vardi (Eds.), FORTE, in: LNCS, vol. 2529, Springer, 2002, pp. 243–259.
[7] J.C.M. Baeten, A brief history of process algebra, Theoretical Computer Science 335 (2–3) (2005) 131–146.
[8] J.C.M. Baeten, C.A. Middelburg, Process Algebra with Timing, Springer-Verlag New York, Inc, Secaucus, NJ, USA, 2002.
[9] Gérard Berry, The foundations of esterel, in: Gordon D. Plotkin, Colin Stirling, Mads Tofte (Eds.), Proof, Language, and Interaction, The MIT Press, 2000, pp. 425–454.
[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, Yunshan Zhu, Bounded model checking, Advances in Computers 58 (2003) 118–149.
[11] M.M. Bonsangue, D. Clarke, A. Silva, Automata for context-dependent connectors, in: J. Field, V.T. Vasconcelos (Eds.), COORDINATION, in: LNCS, vol. 5521, Springer, 2009, pp. 184–203.
[12] Luca Cardelli, Real time agents, in: Mogens Nielsen, Erik Meineche Schmidt (Eds.), ICALP, in: Lecture Notes in Computer Science, vol. 140, Springer, 1982, pp. 94–106.
[13] E.M. Clarke, A. Biere, R. Raimi, Y. Zhu, Bounded model checking using satisfiability solving, Formal Methods in System Design 19 (1) (2001) 7–34.
[14] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, Journal of the ACM 50 (5) (2003) 752–794.
[15] David Costa, Formal Models for Component Connectors, Ph.D. Thesis, Vrije Universiteit Amsterdam, 2010.
[16] Eclipse Coordination Tools, http://reo.project.cwi.nl/.
[17] Wan Fokkink, Introduction to Process Algebra, Springer-Verlag New York, Inc, Secaucus, NJ, USA, 2000.
[18] R. Hähnle, Short CNF in finitely-valued logics, in: H.J. Komorowski, Z.W. Ras (Eds.), ISMIS, in: LNCS, vol. 689, Springer, 1993, pp. 49–58.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, K.L. McMillan, Abstractions from proofs, in: N.D. Jones, X. Leroy (Eds.), POPL, ACM, 2004, pp. 232–244.
[20] R. Jhala, K.L. McMillan, Interpolant-based transition relation approximation, in: K. Etessami, S.K. Rajamani (Eds.), CAV, in: LNCS, vol. 3576, Springer, 2005, pp. 39–51.
[21] Stephanie Kemper, SAT-based verification for timed component connectors, Electronic Notes in Theoretical Computer Science 255 (2009) 103–118.
[22] Stephanie Kemper, A. Platzer, SAT-based abstraction refinement for real-time systems, Electronic Notes in Theoretical Computer Science 182 (2007) 107–122.
[23] Kim Guldstrand Larsen, Robin Milner, Verifying a protocol using relativized bisimulation, in: Thomas Ottmann (Ed.), ICALP, in: Lecture Notes in Computer Science, vol. 267, Springer, 1987, pp. 126–135.
[24] The MathSAT 4 SMT solver, http://mathsat4.disi.unitn.it.
[25] K.L. McMillan, An interpolating theorem prover, in: K. Jensen, A. Podelski (Eds.), TACAS, in: LNCS, vol. 2988, Springer, 2004, pp. 16–30.
[26] R. Milner, Communication and Concurrency, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
[27] Robin Milner, Calculi for synchrony and asynchrony, Theoretical Computer Science 25 (1983) 267–310.
[28] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: DAC, ACM, 2001, pp. 530–535.
[29] Vaughan R. Pratt, Modeling concurrency with geometry, in: POPL, 1991, pp. 311–322.
[30] G.M. Reed, A.W. Roscoe, A timed model for communicating sequential processes, Theoretical Computer Science 58 (1988) 249–261.
[31] Stavros Tripakis, Verifying progress in timed systems, in: Joost-Pieter Katoen (Ed.), ARTS, in: Lecture Notes in Computer Science, vol. 1601, Springer, 1999, pp. 299–314.
[32] W. Yi, CCS + Time = an interleaving model for real time systems, in: J.L. Albert, B. Monien, M. Rodríguez-Artalejo (Eds.), ICALP, in: LNCS, vol. 510, Springer, 1991, pp. 217–228.