Spatio-Temporal Model-Checking of Cyber-Physical Systems Using Graph Queries

Hojat Khosrowjerdi[1](✉), Hamed Nemati[2], and Karl Meinke[1]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
hojatk@kth.se

[2] Helmholtz Center for Information Security (CISPA), Saarbrücken, Germany

Abstract. We explore the application of *graph database technology* to *spatio-temporal model checking* of cooperating cyber-physical systems-of- systems such as vehicle platoons. We present a translation of *spatio-temporal automata* (STA) and the *spatio-temporal logic* STAL to semantically equivalent *property graphs* and *graph queries* respectively. We prove a sound reduction of the spatio-temporal verification problem to graph database query solving. The practicability and efficiency of this approach is evaluated by introducing *NeoMC*, a prototype implementation of our explicit model checking approach based on Neo4j. To evaluate NeoMC we consider case studies of verifying vehicle platooning models. Our evaluation demonstrates the effectiveness of our approach in terms of execution time and counterexample detection.

1 Introduction

In cooperating cyber-physical systems-of-systems (CO-CPS) such as vehicle platoons, with hard real-time and spatial requirements, even the slightest failure of a service may be catastrophic and endanger lives. Severe consequences of such failures reinforce the need for developing rigorous analysis techniques to increase the safety of CO-CPS. Recently, spatio-temporal verification [1–3] appears as a promising technique to verify advanced autonomous services that incorporate temporal and physical features to safely interact with the environment. The high complexity of such systems, however, makes scalable static analysis computationally challenging in practice. Therefore, to make safety certification practical, the analysis of CO-CPS also needs dynamic techniques for ensuring correct and safe functionality, such as model-based and learning-based testing.

There has been a large body of work related to specifying and verifying real-time systems. Examples include Timed Automata [4] and Duration Calculus [5]. None of these formalisms, however, are sufficient for problems with spatial requirements. We propose a new model checking approach based on *spatio-temporal automaton logic* (STAL) [1] to analyze systems having both temporal and spatial characteristics, e.g. CO-CPS. While several other works have also addressed this problem [2,3,6,7], a distinguishing feature of our approach is

© Springer Nature Switzerland AG 2020
W. Ahrendt and H. Wehrheim (Eds.): TAP 2020, LNCS 12165, pp. 59–79, 2020.
https://doi.org/10.1007/978-3-030-50995-8_4

the adoption of *graph databases* and *graph queries* [8] for model checking. This enables us to gain advantages in terms of *counterexample detection*, *analysis time* and *memory consumption*.

In [1], STAL was introduced as a requirements modeling language for systems of distributed dynamic objects, such as autonomous vehicles. (See Sect. 4). STAL is based on a restricted subset of first-order linear temporal logic (FOLTL) with dedicated real-valued spatial functions. To avoid undecidability problems associated with infinite state spaces, STAL semantics is based on finite *spatio-temporal automata* (STA) models. These properties of STAL make it a potentially practicable logic for modeling safety requirements on CO-CPS such as collision avoidance and safety envelopes.

A finite state STA can be machine learned (ML) using techniques of finite automaton learning [9]. By combining ML with the model checking methods for STAL presented here, we can implement learning-based testing (LBT) [10,11] as a dynamical safety assurance method for CO-CPS.

However, there are several problems with using off-the-shelf model checkers to check STAL properties. Most existing model checkers do not support FOLTL which is essential to verify spatial properties directly on the model and without manually crafting new model features. Additionally, ML-generated STA models are large, flat and unstructured. This prevents many model checkers from optimizing the search computation, or using compact internal representations of the state space. We try to address these issues and show how the STAL model checking problem can be soundly implemented by graph database search (Sect. 6). We show the practicability of this approach (Sect. 7) by developing an explicit state model checker, called **NeoMC**, using the graph database *Neo4j* and its declarative query language *Cypher* [12,13] (Sect. 5). We apply our model checker to large case studies and report the results in Sect. 7. Our benchmarking results show the practicability and effectiveness of our approach in terms of counterexample detection and execution time. Most importantly, NeoMC has enabled us to model check requirements and models that are otherwise not efficiently structured to be verified using other available model checkers.

2 Overview

Figure 1 depicts an application of STAL model checking in a dynamical safety assurance toolchain based on learning-based testing (LBT) [11]. LBT uses active automaton learning [14] to reverse engineer a state machine model of an SUT that can be guaranteed to be both complete and correct. This model is learned iteratively as a sequence of increasingly larger and more accurate models $M_0, M_1, \ldots$, by alternating between active learning, model checking and equivalence checking queries. After each iteration, the current model M_i is checked to find potential discrepancies with respect to functional requirements on the SUT. If each M_i is an STA then these requirements can be spatio-temporal requirements formalized in STAL [1].

In LBT, requirements testing is implemented by the model checker to evaluate whether an inferred model M_i complies with the given requirements.

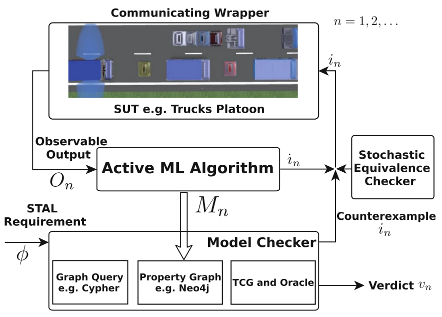

Fig. 1. LBT using graph queries.

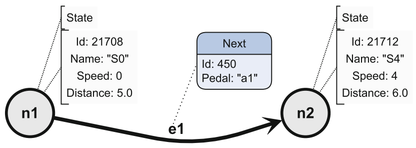

Fig. 2. Example property graph showing part of a platoon state machine.

In this way, the model checker functions as the test oracle to generate pass or fail verdicts.

One of our main goals in implementing a dedicated model checker for STAL was to try to improve the performance of model checking for LBT. For this purpose we can represent an STA model M as a property graph data model $\mathcal{G}_M$ in Neo4j [12] and we can model a STAL requirement ϕ using the high level graph query language Cypher [13]. We can then use graph queries to search for potential counterexamples in $\mathcal{G}_M$ that falsify ϕ in M. Thus we can reduce model checking to a query matching problem.

Neo4j is a high-performance graph database that stores data in graphs (represented as a key-value database) rather than in tables. Using graph representation, Neo4j is able to capture the inherent graph structure of data appearing in applications such as geographic information systems (GIS), where data paths and navigational patterns are important [15]. The data processing engine in Neo4j utilizes *index-free adjacency* [16]. In this approach each node keeps micro-indexing of its adjacent nodes, thus reducing the query response time and making it independent of the total graph size. Neo4j is a fully transactional directed graph database and allows assigning attributes (key/value pairs) to nodes (vertices) and relationship (edges). It can efficiently handle connected data and supports various data-types (e.g. floating point, integer, strings). This makes Neo4j well suited for storing various types of automata models including STA, since nodes can represent states and edges naturally represent transitions.

Cypher [12,13] is a graph query language capable of specifying graph patterns between nodes that may span over arbitrary-length paths. Cypher is a declarative language allowing users to express queries without a deeper understanding of the underlying system. However, it is expressive enough to support complex query patterns related to graph analytics [17]. A Cypher query takes a property graph as the input and performs various computations on it, returning a table of values.

3 Preliminaries

A *Property Graph Model* (PGM) is a directed labeled graph in which nodes and edges have attributes, also called *properties*. A property is a pair of the form (*key,value*). Values can be basic data types, such as strings and integers, or composite, such as lists, maps and paths.

Let Σ be an alphabet, then Σ^* denotes the set of all strings over the alphabet. We let $K \subset \Sigma^*$ denote a finite set of property keys, and $\mathcal{A} \subseteq \Sigma^*$ denotes a possibly infinite set of variable names. We define node labels L and edge types T as countably infinite sets of strings from Σ^*. Also, V is the set of values and it contains:

- Node and edge identifiers.
- Base types: integers $\mathbb{Z}$, real numbers $\mathbb{R}$, and strings Σ^*.
- Booleans: $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$.
- **null**: denoting an undefined value.
- Lists: an empty or non-empty list of values $list(v_1, v_2, ..., v_m)$.
- Maps: an empty or non-empty set of $(key, value)$ pairs.
- Paths: a sequence of node and edge identifiers $(n_0, e_0, n_1, ..., n_{k-1}, e_{k-1}, n_k)$.

A *property graph* is an 8-tuple $\mathcal{G} = (N, E, L, T, \lambda, Lab, Typ, P_{node}, P_{edge})$ consisting of a set $N \subseteq \Sigma^*$ of *node identifiers*, and a set $E \subseteq \Sigma^*$ of *directed edge identifiers*. We associate a *label set* to each node by the function $Lab : N \rightarrow 2^L$. Similarly, we assign a *type* $t \in T$ (or possibly **null**) to each edge by the function $Typ : E \rightarrow T \cup \{\mathbf{null}\}$. Furthermore, $\lambda : E \rightarrow N \times N$ is the function which yields the source and the target nodes for a given edge. Note that two edge identifiers may have the same source/target nodes.

By a *property* we mean a pair $p = (k, v) \in K \times V$ consisting of a key k and a value v. Then $P_{node} : N \times K \rightarrow V \cup \{\mathbf{null}\}$ is the *property labelling* function for nodes which maps each node n and key k to the corresponding value (that could be **null**). Similarly, $P_{edge} : E \times K \rightarrow V \cup \{\mathbf{null}\}$ is the *property labelling* function for edges which maps each edge e and key k to the corresponding value (possibly **null**).

Example 1. Figure 2 exemplifies a property graph. In this example, there are two nodes and one edge, namely n_1, n_2 and e_1. The node's label is "`State`" and the edge type is "`Next`". Node property keys are "`Id`", "`Name`", "`Speed`" and "`Distance`" and the edge property keys are "`Id`" and "`Pedal`". The value of the each property is given below.

$L = \{\texttt{State}\}$		
$N = \{n_1, n_2\}$	$\mathit{Lab}(n_1) = \{\texttt{State}\}$	$\mathit{Lab}(n_2) = \{\texttt{State}\}$
	$P_{node}(n_1, \texttt{Id}) = 21708$	$P_{node}(n_2, \texttt{Id}) = 21712$
	$P_{node}(n_1, \texttt{Name}) =$ "$S0$"	$P_{node}(n_2, \texttt{Name}) =$ "$S4$"
	$P_{node}(n_1, \texttt{Speed}) = 0$	$P_{node}(n_2, \texttt{Speed}) = 4$
	$P_{node}(n_1, \texttt{Distance}) = 5.0$	$P_{node}(n_2, \texttt{Distance}) = 6.0$
$T = \{\texttt{Next}\}$		
$E = \{e_1\}$	$\lambda(e_1) = (n_1, n_2)$	$\mathit{Typ}(e_1) = \{\texttt{Next}\}$
	$P_{edge}(e_1, \texttt{Id}) = 450$	$P_{edge}(e_1, \texttt{Pedal}) =$ "$a1$"

Having defined a property graph, we define a *path* in such a graph as follows. Let $n_i, n_k \in N$ and $e_j \in E$ be node and edge identifiers of a property graph $\mathcal{G}$. A *path* $\overline{w}$ from n_i to n_k denoted by $\overline{w} = (n_i \rightarrow^* n_k)$ is a finite sequence of nodes and edges $(n_i e_i n_{i+1} ... n_{k-1} e_{k-1} n_k)$ such that $\forall i \leq j < k : \lambda(e_j) = (n_j, n_{j+1})$.

We use $\prod_{type}^{size}(\mathcal{G})$ to denote the set of all paths in $\mathcal{G}$ of length *size* and a specific *type* in the input *model*. Then $\prod(\mathcal{G})$ denotes the set of all paths in $\mathcal{G}$ (of any finite length). If $\overline{w}_1 = (n_0 e_0 n_1 ... e_{k-1} n_k)$ and $\overline{w}_2 = (n_k e_k ... e_{i-1} n_i)$, then we denote the order-preserving concatenation of $\overline{w}_1$ and $\overline{w}_2$ by $\overline{w}_1 . \overline{w}_2 = (n_0 e_0 n_1 ... e_{k-1} n_k e_k ... e_{i-1} n_i)$.

In a graph database, to create, read and update property graphs, graph queries are executed. A query $\mu \in \mathcal{Q}$ takes a graph $\mathcal{G}$ as an input and returns a table $\mathbb{t} \in \mathbb{T}$. This table provides parameter bindings that match the query to a solution in the graph.

Let $u = \{a_1 : v_1, ..., a_n : v_n\}$ be an assignment(record) from variable names $\{a_1, ..., a_n\} \subset \mathcal{A}$ to values $\{v_1, ..., v_n\} \subset V$, and $dom(u)$ denotes the domain of u, i.e. $dom(u) = \{a_1, ..., a_n\}$. A table $\mathbb{t} \in \mathbb{T}$ is a multiset (bag) of assignments that have a common domain $\mathcal{A}$. In other words, tables are partial mappings from names (columns) to values, without any specific ordering.

4 Spatio-Temporal Automaton Logic (STAL)

In [1] we presented a modal logic STAL suitable for describing the spatio-temporal behavior of a spatially distributed dynamical system of objects, such as autonomous vehicles or drones. Such systems have many dynamically changing properties such as object locations, distances and velocities. These properties may be expressed using relative or absolute coordinates. Following classical Newtonian physics, such properties are usually resolved into their vector components along 1, 2 or 3 spatial dimensions as appropriate.

Formally, STAL is a quantifier-free fragment[1] of first-order linear temporal logic (FOLTL). The semantics of STAL can therefore be defined in a similar

[1] To ensure decidability, STAL is syntactically restricted so that quantification over data types is not allowed.

way to FOLTL, in terms of a *spatio-temporal automaton* (STA) that interprets the spatial operators of the logic. A key requirement for learning-based testing (LBT) [11] is that spatio-temporal automata are amenable to machine learning in finite time in much the same way as finite automata [9]. Successful LBT also requires the existence of a decidable model checking problem and an efficient model checking algorithm such as the one presented in this work.

STAL can be used to describe a dynamically changing environment of spatially distributed objects by relativising spatio-temporal measurements to a distinguished object called the *ego object*. The ego object provides an origin and point of reference in each dimension for every *relative spatio-temporal property* (e.g. relative distance). Thus FOLTL provides an implicit temporal reference to *now*, while the ego object provides the corresponding spatial reference to *here*. Furthermore, by supporting the measurement of bounded relative properties, STA allow us to avoid infinite state automata models in many practical situations. This means that both machine learning and model checking of spatio-temporal automata can be achieved in finite time using regular inference and explicit state space search methods.

Taking the common case of 2 orthogonal spatial dimensions, the x and y axes, we can define a 2-dimensional[2] STA to be the following algebraic structure:

$$\begin{aligned}A = (&\Sigma, Q, Obj, q_0, egoObj,\\ &\delta : \Sigma \times Q \to Q,\ angle : Q \to [0, .., 360),\\ &dist_x,\ dist_y,\ vel_x,\ vel_y : Obj \times Q \to \mathbb{R}).\end{aligned}$$

Here $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ is a finite *input alphabet*, consisting of ordered key-value pairs[3] $p = (k, v) \in K \times V$, $Q = \{q_0, \ldots, q_n\}$ is a finite set of states, $Obj = \{o_1, \ldots, o_k\}$ is a finite set of objects, $q_0 \in Q$ is the distinguished initial state, $egoObj \in Obj$ is the distinguished ego object, $\delta : \Sigma \times Q \to Q$ is the state transition function, $angle : Q \to [0, .., 360)$ gives the *ego object orientation* relative to the x, y axes, $dist_x, dist_y : Obj \times Q \to \mathbb{R}$ are the *relative object distance* functions along the x, y axes measured from the ego object for each state, and $vel_x, vel_y : Obj \times Q \to \mathbb{R}$ are the *absolute object velocity* functions[4] measured along the x, y axes for each state.

Example 2. Figure 3 shows a simple STA consisting of three states q_0, q_1, q_2. It describes the movements of two vehicles in a platoon, namely *leader* and *follower*. The *leader* is controlled by a driver using gas and brake pedals. Then, the input alphabet Σ is a set consisting of $\sigma_0 = (Pedal, \text{"}gas\text{"})$ and $\sigma_1 = (Pedal, \text{"}brake\text{"})$. Both vehicles are driving along the x axis. The *follower* object, which is the ego object, tries to adapt its distance and speed to the *leader* object motion. This example STA is two dimensional and all distances are measured along x and y axes with respect to *follower* as the ego object. For all states of the automaton,

[2] This definition clearly generalises to the n-dimensional case.

[3] denoted as "(*key,value*)".

[4] Note: we can derive relative velocity from absolute velocity, and both measurements are always bounded in practise.

the angle of the ego vehicle, the inter-vehicle distance along the y axis of the ego object and the absolute vehicle velocities in the y dimension are zero. The transition function δ is defined as follows. Initially (in state q_0), the *leader* is 50 meters ahead of the *follower* along its x axis. This distance will be reduced to 20 and 10 meters if the *leader* accelerates (*Pedal, gas*) or brakes (*Pedal, brake*) respectively. If the driver pushes the gas pedal, the speed of the *leader* increases from zero to 50 km/h along the x axis. Should the brake pedal be pressed, the *leader* speed drops to 30 km/h. At the same time, the ego vehicle tries to follow this speed pattern at 48 km/h and 29 km/h.

The formal syntax of STAL is summarized in Fig. 4. In this Figure *exp*, exp_1 and exp_2 are arithmetic expressions, and ϕ, ϕ_1 and ϕ_2 are arbitrary STAL formulas. Let $\mathcal{S}$ denote the set of all STAL formulas.

$\sigma_0 = (Pedal, \text{“}gas\text{”}), \sigma_1 = (Pedal, \text{“}brake\text{”})$ $Q = \{q_0, q_1, q_2\}$ $egoObj = follower$	$\Sigma = \{\sigma_0, \sigma_1\}$ $Obj = \{leader, follower\}$
$\delta(\sigma_0, q_0) = q_1, \delta(\sigma_1, q_0) = q_0$ $\delta(\sigma_0, q_2) = q_1, \delta(\sigma_1, q_2) = q_2$	$\delta(\sigma_0, q_1) = q_1, \delta(\sigma_1, q_1) = q_2$
$angle(q_0) = 0, angle(q_1) = 0, angle(q_2) = 0$	
$dist_x(leader, q_0) = 50, dist_x(leader, q_1) = 20,$ $dist_y(leader, q_0) = 0, dist_y(leader, q_1) = 0,$	$dist_x(leader, q_2) = 10$ $dist_y(leader, q_2) = 0$
$vel_x(leader, q_0) = 0, vel_x(leader, q_1) = 50,$ $vel_y(leader, q_0) = 0, vel_y(leader, q_1) = 0,$	$vel_x(leader, q_2) = 30$ $vel_y(leader, q_2) = 0$
$vel_x(follower, q_0) = 0, vel_x(follower, q_1) = 48,$ $vel_y(follower, q_0) = 0, vel_y(follower, q_1) = 0,$	$vel_x(follower, q_2) = 29$ $vel_y(follower, q_2) = 0$

Fig. 3. An example of an STA.

$exp ::= \ c \in \mathbb{R} \ \mid \texttt{next(}exp\texttt{)}$
$\quad \mid \texttt{Distance}_x\texttt{(o)} \mid \texttt{Distance}_y\texttt{(o)} \mid \texttt{Speed}_x\texttt{(o)} \mid \texttt{Speed}_y\texttt{(o)} \mid \ \texttt{Angle}$; where
$\texttt{o} \in Obj$
$\quad \mid exp_1 + exp_2 \mid exp_1 - exp_2 \mid exp_1 * exp_2 \mid exp_1 / exp_2$
$\phi ::= \ \texttt{input} \ = \sigma$, (for $\sigma \ \in \Sigma$)
$\quad \mid (exp_1 < exp_2) \ \mid (exp_1 > exp_2) \ \mid (exp_1 \leq exp_2) \ \mid \ (exp_1 \geq exp_2) \mid (exp_1 \neq exp_2)$
$\quad \mid \ \neg(\phi) \mid (\phi_1 \ \wedge \ \phi_2) \mid (\phi_1 \ \vee \ \phi_2) \mid (\phi_1 \rightarrow \phi_2) \mid \ X(\phi) \mid F(\phi) \mid G(\phi)$

Fig. 4. Syntax of STAL.

For a given object $o \in Obj$, a STAL expression *exp* is either a floating point constant c, a distance expression $\texttt{Distance}_x\texttt{(o)}$ or $\texttt{Distance}_y\texttt{(o)}$, a speed expression $\texttt{Speed}_x(o)$ or $\texttt{Speed}_y(o)$, an angle expression `Angle` or a binary arithmetic operation ($+$, $-$, $*$, $/$) applied to two subexpressions exp_1, exp_2.

An *atomic STAL formula* is either an input expression ($\texttt{input} = \sigma$) for $\sigma \in \Sigma$ or a pair of arithmetic expressions connected by an arithmetic relation ($<$, $>$, $\leq$,

$\geq$). A *compound STAL formula* ϕ may be built up from subformulas by means of boolean operations (i.e. $\neg$, $\wedge$, $\vee$, or $\rightarrow$), and linear temporal operators (i.e. next X, eventually F, or always G).

The semantics of STAL is defined in Fig. 5. To define the satisfiability relation $\models$ we write $w = (\alpha_0, \alpha_1, ...) \in \prod^{\infty}(A)$ to denote an infinite path in A and we write its suffixes as $w^i = (\alpha_i, \alpha_{i+1}, ...)$ for $i \geq 0$. Note that a path in a spatio-temporal automaton is a sequence of input (σ) and state (q) pairs, i.e. $\alpha_i = (\sigma_i, q_i)$. We use ($\alpha_i \rightarrow^* \alpha_k$) to indicate a path from q_i to q_k if $\forall i \leq j < k : q_{j+1} = \delta(\sigma_j, n_j)$ and the last input $\sigma_k = \epsilon$ is an empty string.

$$
\begin{aligned}
&[\![c]\!]_\alpha = c\\
&[\![Distance_x(o_i)]\!]_\alpha = dist_x(o_i, q)\\
&[\![Distance_y(o_i)]\!]_\alpha = dist_y(o_i, q)\\
&[\![Angle]\!]_\alpha = angle(q)\\
&[\![Speed_x(o_i)]\!]_\alpha = vel_x(o_i, q)\\
&[\![Speed_y(o_i)]\!]_\alpha = vel_y(o_i, q)\\
&[\![next(exp)]\!]_{\alpha_i} = [\![exp]\!]_{\alpha_{i+1}}\\
&[\![exp_1 \ \mathbf{bop} \ exp_2]\!]_\alpha =\\
&\quad [\![exp_1]\!]_\alpha \ \mathbf{bop} \ [\![exp_2]\!]_\alpha\\
&[\![\mathtt{input} = \sigma]\!]_{\alpha_i} = \begin{cases} \mathbf{true} & if \sigma = \sigma_i \\ \mathbf{false} & if \sigma \neq \sigma_i \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&A, w \models (\mathtt{input} = \sigma) \iff [\![\mathtt{input} = \sigma]\!]_{\alpha_0}\\
&A, w \models (exp_1 \, \mathbf{bop} \, exp_2) \iff\\
&\quad [\![exp_1 \, \mathbf{bop} \ exp_2]\!]_{\alpha_0}\\
&A, w \models \neg\phi \iff A, w \not\models \phi\\
&A, w \models (\phi_1 \wedge \phi_2) \iff A, w \models \phi_1 \wedge A, w \models \phi_2\\
&A, w \models (\phi_1 \vee \phi_2) \iff A, w \models \phi_1 \vee A, w \models \phi_2\\
&A, w \models (\phi_1 \rightarrow \phi_2) \iff A, w \not\models \phi_1 \vee A, w \models \phi_2\\
&A, w \models X(\phi) \iff A, w^1 \models \phi\\
&A, w \models G(\phi) \iff \forall i \in \mathbb{N} : A, w^i \models \phi\\
&A, w \models F(\phi) \iff \exists i \in \mathbb{N} : A, w^i \models \phi
\end{aligned}
$$

Fig. 5. STAL semantics and its satisfiability relation over a path w of an STA A.

$exp ::= v \in V \mid a \in \mathcal{A} \mid f(exp), f \in F$ $\quad \mid exp.k \ \mid \{\mathcal{P}\}$	values/variables
$mid[exp]$	maps/lists (for $k \in K$)
$\quad \mid exp_1$ `OR` $exp_2 \mid exp_1$ `AND` $exp_2 \mid$ `NOT` exp	logic
$\quad \mid exp_1 \ \mathbf{bop} \ exp_2$	comparison
$pattern ::= \pi \mid a = \pi$ (for $a \in \mathcal{A}$)	$\mathcal{L} \ ::= :l \ \mid :l\mathcal{L}$ (for $l \in L$)
$\pi ::= \chi \mid \chi\rho\pi$	$\mathcal{P} \ ::= k : exp \mid k : exp, \mathcal{P} \mid \mathbf{null}$
$\chi ::= (a \ \mathcal{L}? \ \{\mathcal{P}\}?)$	$\mathcal{T} \ ::= :t \ \mid \mathcal{T} \mid \mathbf{null}$ (for $t \in T$)
$\rho ::= \text{-}[a \ \mathcal{T}? \ \mathcal{I}? \ \{\mathcal{P}\}?] \rightarrow$	$len ::= d \mid d_1.. \mid ..d_2 \mid d_1..d_2$
$\quad \mid \leftarrow [a \ \mathcal{T}? \ \mathcal{I}? \ \{\mathcal{P}\}?]\text{-}$	$\quad \mid \mathbf{null}$ (for $d, d_1, d_2 \in \mathbb{N}$)
	$\mathcal{I} ::= *len?$
$ret ::= exp$	
$query ::=$ `MATCH` $pattern$ `WHERE` exp `RETURN` ret	

Fig. 6. Core syntax of Cypher for model checking

5 Cypher Syntax and Semantics

In this section we present a subset of the Cypher language which is sufficient for interpreting STAL formulas. Cypher includes *expressions*, *patterns*, *clauses*, and *queries*, which allow it to represent a data model represented as *values*, *graphs* and *tables*. The syntax of Cypher is depicted in Fig. 6. We present keywords in blue. The main concepts in Cypher are the notions of "pattern" and "pattern matching". The underlying data set for a query in Cypher is a property graph and the response is a table providing bindings for all query parameters representing solutions found in the property graph.

The `MATCH` clause denotes a matching function from tables to tables and may introduce new rows (synonymous with records) with bindings of the matched instances of the pattern in the queried graph. Similar to other query languages, the `WHERE` clause in Cypher filters the results of this matching based on the valid filter predicates. These predicates can be defined based on the properties of query elements. For example, `Match (n) WHERE n.k = value` is a query to match all nodes in a graph that satisfy the attribute restriction $k = value$ for a property $p : (k, value)$ of a node n. The binary operations, **bop**, are the standard ones and we use them to express the relation between two properties or properties and literals. The keyword `RETURN` expresses the projection of the result.

For model checking purposes, Cypher expressions are used in the `WHERE` clause to apply predicate conditions and filter search results. They also appear in the `RETURN` statement, e.g., to define how a counterexample should be structured and returned properly. Expressions can also be used in patterns to parameterize node and edge properties during a pattern matching search.

$$[\![v]\!]_{\mathcal{G},u} = v,\ v \in V \qquad [\![a]\!]_{\mathcal{G},u} = u(a), a \in dom(u)$$

$$[\![f(exp_1, ..., exp_m)]\!]_{\mathcal{G},u} = f([\![exp_1]\!]_{\mathcal{G},u}, ..., [\![exp_m]\!]_{\mathcal{G},u})$$

$$\forall k, k_i \in K.$$

$$[\![exp.k]\!]_{\alpha_i} = \begin{cases} P_{node}([\![exp]\!]_{\mathcal{G},u}, k) & if\ [\![exp]\!]_{\mathcal{G},u} \in N \\ P_{edge}([\![exp]\!]_{\mathcal{G},u}, k) & if\ [\![exp]\!]_{\mathcal{G},u} \in E \\ v_i & if\ [\![exp]\!]_{\mathcal{G},u} = map((k_1, v_1), \ldots, (k_i, v_i), \ldots) \\ & \quad and\ k = k_i \\ \mathbf{null} & if\ [\![exp]\!]_{\mathcal{G},u} = map((k_1, v_1), \ldots, (k_i, v_i), \ldots) \\ & \quad and\ k \notin \{k_1, \ldots, k_i\} \\ & or\ [\![exp]\!]_{\mathcal{G},u} = \{\} \\ & or\ [\![exp]\!]_{\mathcal{G},u} = \mathbf{null} \end{cases}$$

$$[\![\{k_1 : exp_1, \ldots, k_i : exp_i\}]\!]_{\mathcal{G},u} = map((k_1, [\![exp_1]\!]_{\mathcal{G},u}), \ldots, (k_i, [\![exp_i]\!]_{\mathcal{G},u}))$$

$$[\![\texttt{NOT}\,exp]\!]_{\mathcal{G},u} = \begin{cases} true & if\ [\![exp\ ]\!]_{\mathcal{G},u} = false \\ false & if\ [\![exp\ ]\!]_{\mathcal{G},u} = true \\ \mathbf{null} & if\ [\![exp\ ]\!]_{\mathcal{G},u} = \mathbf{null} \end{cases}$$

$$[\![exp_1\ \texttt{AND}\ exp_2]\!]_{\mathcal{G},u} = \begin{cases} true & if\ [\![exp_1]\!]_{\mathcal{G},u} = [\![exp_2]\!]_{\mathcal{G},u} = true \\ false & if\ [\![exp_1]\!]_{\mathcal{G},u} = false\ or\ [\![exp_2]\!]_{\mathcal{G},u} = false \\ \mathbf{null} & otherwise \end{cases}$$

$$[\![exp_1\ \texttt{OR}\ exp_2]\!]_{\mathcal{G},u} = \begin{cases} true & if\ [\![exp_1]\!]_{\mathcal{G},u} = true\ or\ [\![exp_2]\!]_{\mathcal{G},u} = true \\ false & if\ [\![exp_1]\!]_{\mathcal{G},u} = [\![exp_2]\!]_{\mathcal{G},u} = false \\ \mathbf{null} & otherwise \end{cases}$$

Fig. 7. Cypher expression semantics.

5.1 Cypher Patterns

Syntax. Cypher supports three types of patterns: *node* (χ), *edge* (ρ) and *path* (π) patterns. In a path-based temporal logic such as FOLTL, path patterns can be used to describe a counterexample as a path to a node or group of nodes where some desired properties are violated. Patterns can be recursively defined using the derivation rules in Fig. 6. In this figure f is any m-ary function in F from values to values, e.g., `All` and `Any`, and $exp.k$ returns a pair from a map with a matching key k, i.e. $v_i = map((k_1, v_1), \dots, (k_i, v_i), \dots).k_i$. We use "?" to denote optional (or "nullable") types, for which **null** represents missing (or None) values. Also "$*$" denotes a range $[d_1, d_2]$ with $d_1, d_2 \in \mathbb{N}$ specified by the optional len for the edge pattern ρ. The range is equal to $[1, \infty]$ if len is **null** or $[d, d]$, $[d_1, \infty]$, $[1, d_2]$, $[d_1, d_2]$ if other derivation rules are applied, respectively.

Definition 1 (Node Pattern). *A node pattern χ has the form $\chi = (a\ \mathcal{L}?\ \{\mathcal{P}\}?)$ where $a \in N$ is a node name, $\mathcal{L}$ is an optional finite set of node labels, and $\{\mathcal{P}\}$ is an optional partial mapping from property keys k to expressions exp. For example (x), $(x\!:\!State)$ and $(x\{Name\!:\ \text{“S0”}\})$ are node patterns.*

Definition 2 (Edge Pattern). *An edge pattern ρ has the form $\rho = (a\ \mathcal{T}?\ \mathcal{I}?\ \{\mathcal{P}\}?\ dir)$ where $a \in E$ is an edge name, $\mathcal{T}$ is an optional edge type, $\mathcal{I}$ indicates an optional range for the length of the edge between source/target nodes, $\mathcal{P}$ is an optional partial mapping from property keys to expressions and $dir \in \{\rightarrow, \leftarrow\}$ indicates the direction.*

Definition 3 (Path Pattern), *A path pattern π is a concatenation of node and edge patterns of the form $\chi_1\rho_1\chi_2\rho_2 ... \chi_n$.*

Henceforth we write $\pi = (n_1)\text{-}[e]\!\!\rightarrow\!(n_2)$ where $n_1, n_2 \in N$ and $e \in E$, instead of $\pi = \chi_1\rho\chi_2$ to denote the syntactic category pattern defined in Fig. 6. Using this notation, patterns can encode paths as nodes and edges with arrows between them to indicate the direction of a transition.

Semantics. The semantics of a pattern is the set of nodes, edges or paths which satisfy its conditions. For example, the semantics of a path pattern π is the path value $[\![\pi]\!]_{\mathcal{G},u} \in V$. Figure 7 shows the semantics of cypher expressions where the semantics of an expression exp is a value $[\![exp]\!]_{\mathcal{G},u} \in V$ determined by $\mathcal{G}$ and u. For example, for a constant $v \in V$, a variable name $a \in \mathcal{A}$ and an m-array function $f \in F$, the semantic values are $[\![v]\!]_{\mathcal{G},u} = v$, $[\![a]\!]_{\mathcal{G},u} = u(a)$ and $[\![f(exp_1, ..., exp_m)]\!]_{\mathcal{G},u} = f([\![exp_1]\!]_{\mathcal{G},u}, ..., [\![exp_m]\!]_{\mathcal{G},u})$ respectively. The complete semantics is given in [18].

Definition 4 (Path Value). *A path value for a pattern π in $\mathcal{G}$ given the assignment u which provides name bindings for π and $\mathcal{G}$, is a set of paths $\overline{w}$ in $\mathcal{G}$ such that, $[\![\pi]\!]_{\mathcal{G},u} = \{\overline{w} \in \prod(\mathcal{G}) \mid (\overline{w}, \mathcal{G}, u) \models \pi\}$.*

For example, the pattern $\pi = (n)\text{-}[e]\!\!\rightarrow\!(m)$ indicates a set of paths $\{(n_0e_0n_1) | n_0, n_1 \in N, e_0 \in E\}$ of length one in the graph $\mathcal{G}$ using the assignment

$u = (n : n_0, e : e_0, m : n_1)$, and n_0, e_0, n_1 are any node and edge identifiers in $\mathcal{G}$ with the relation $\lambda(e_0) = (n_0, n_1)$. Note that n_0, n_1 and e_0 can be any nodes and edges within the graph that satisfy this edge pattern.

Property 1. Let ρ be an edge pattern $(a\ \mathcal{T}?\ \mathcal{I}?\ \{\mathcal{P}\}?\ dir)$, χ be a node pattern $(a\ \mathcal{L}?\ \{\mathcal{P}\}?)$, $d_1 \le i \le d_2$ and $j \in \{1, ..., i\}$, then a path $\overline{w}$ in a graph $\mathcal{G}$ satisfies a pattern π (i.e., $(\overline{w}, \mathcal{G}, u) \models \pi$) if:

$$(n, \mathcal{G}, u) \models \chi \Leftrightarrow \begin{cases} u(a) = n \\ \mathcal{L} \subseteq Lab(n) \\ \forall k \in K. \\ \quad [\![P_{node}(n,k) = \{\mathcal{P}\}.k]\!]_{\mathcal{G},u} = true \end{cases} \quad \text{and} \quad (n_1 ... e_i n_{i+1}.\overline{w}, \mathcal{G}, u) \models \chi\rho\pi \Leftrightarrow \begin{cases} (n_1, \mathcal{G}, u) \models \chi \\ (\overline{w}, \mathcal{G}, u) \models \pi \\ u(a) = list(e_1, ..., e_i) \\ Typ(e_j) \in \mathcal{T} \\ \forall k \in K. \\ \quad [\![P_{edge}(e_j, k) = \{\mathcal{P}\}.k]\!]_{\mathcal{G},u} = true \\ \lambda(e_j) \in \begin{cases} \{(n_j, n_{j+1})\};\ \textit{if dir is } \rightarrow \\ \{(n_{j+1}, n_j)\};\ \textit{if dir is } \leftarrow \end{cases} \end{cases}$$

Example 3. Take the property graph $\mathcal{G}$ from Fig. 2 and assume a Cypher pattern $(x\{Name: \text{“}S0\text{”}\})$–[y{Pedal:“a1”}]→$(z)$, which is equivalent to:

$$\pi = \overbrace{(x\ \textbf{null}\ \{Name\!: \text{“}S0\text{”}\})}^{\chi_1}\overbrace{(y\ \textbf{null}\ \textbf{null}\ \{Pedal\!: \text{“}a1\text{”}\} \rightarrow)}^{\rho}\overbrace{(z\ \textbf{null}\ \textbf{null})}^{\chi_2}$$

where χ_1, χ_2 and ρ are the node and edge patterns. Say $u = (x\!:\!n_1, y\!:\!e_1, z\!:\!n_2)$, then one can show that $(n_1, \mathcal{G}, u) \models \chi_1$, $(e_1, \mathcal{G}, u) \models \rho$ and $(n_2, \mathcal{G}, u) \models \chi_2$. Also, for the path $\overline{w} = (n_1 e_1 n_2)$ it holds that $(\overline{w}, \mathcal{G}, u) \models \pi$ where $\pi = \chi_1 \rho \chi_2$.

Pattern Matching. Central to query satisfiability for a graph database is pattern matching, which is the problem of finding all subgraphs that match a given pattern. A match for a pattern is a function that maps variables to constants such that when applied to the pattern, the result is in the original graph database.

For a node pattern $\chi = (a, \mathcal{L}?, \{\mathcal{P}\}?)$, let $free(\chi) = \{a\}$ be the set of free variables of χ. Similarly, we define free variables of an edge pattern $\rho = (a, \mathcal{T}?, \mathcal{I}?, \mathcal{P}?, dir)$ by $free(\rho) = \{a\}$. Then, the free variables of a path pattern $\pi = \chi_1\rho_1\chi_2\rho_2...\chi_n$ is defined to be the union of all free variables of individual node and edge patterns occurring in it, i.e., $free(\pi) = free(\chi_1) \cup free(\rho_1) \cup ... \cup free(\chi_n)$. We define pattern matching as the function which searches a graph $\mathcal{G}$ to find all paths p that satisfy a pattern π given a variable assignment u from values to variables for the *free variables* of π, i.e. $match(\pi, \mathcal{G}, u) = [\![\pi]\!]_{\mathcal{G},u}$.

For brevity, we drop u and write $match(\pi, \mathcal{G})$ in the sequel. A Cypher matching query $\mu \in \mathcal{Q}$ can be defined as $\mu ::=$ `MATCH` *pattern* `WHERE` *exp* `RETURN` *ret*. The semantics of this query is to call $match(\pi, \mathcal{G})$, apply the predicate conditions of `WHERE` to filter the search results and project the results:

$$[\![\mu]\!]_{\mathcal{G}} = [\![\texttt{MATCH}\ \pi\ \texttt{WHERE}\ exp]\!]_{\mathcal{G}} = \{\overline{w} \in match(\pi, \mathcal{G}) \mid [\![exp]\!]_{\mathcal{G}} = true\}$$

Definition 5. *We say a graph $\mathcal{G}$ satisfies a query μ if and only if there exists a path $\overline{w}$ in $\mathcal{G}$ that is in the semantics of the query. That is:*

$$\mathcal{G} \models \mu \iff \exists \overline{w} \in \prod(\mathcal{G}) \text{ such that } \overline{w} \in [\![\mu]\!]_{\mathcal{G}}.$$

Graph pattern matching is a canonical NP-complete problem. Cypher allows pattern definitions with infinitely many matches (e.g., loops). This makes Cypher impractical in a homomorphism-based semantics [19]. For example, if $\mathcal{G}$ is a graph consisting of a single node n, a single edge e from n to n $(n \rightarrow n)$, then patterns like $\pi = (n)$-[*]$\mapsto(n)$ match $\mathcal{G}$ infinitely many times by iteratively traversing over e if there is no restriction on the number of iterations. Thus, for $i \geq 0$ there exists a match that iterates i times over e in $\mathcal{G}$. Cypher avoids this by using an isomorphism-based semantics [8] to disallow repeating edges while traversing edges in pattern matching. Hence, in the above example, the *match* function only returns two matches, one for $i = 0$ and one for $i = 1$.

6 Spatio-Temporal Model Checking

Spatio-temporal model checking is a variant of classical model checking that combines spatial reasoning with temporal reasoning. Given an STA model A, a STAL formula $\phi \in \mathcal{S}$, and a path $w \in \prod(A)$, model checking analyses whether $A, w \models \neg\phi$ holds. If it does, the path w is returned as a counterexample to ϕ in A. If no counterexample w can be found then model checking returns *true*, i.e., ϕ holds for all possible paths of A.

In traditional explicit state space model checking we usually construct a product automaton from the automaton model and the requirement formula and check this for voidness. By contrast, NeoMC uses *pattern matching* to find counterexamples. For this we translate the requirement into a graph query (*pattern*) and perform pattern matching on a graph model of the automaton to find matches (i.e. counterexamples).

Since STAL is an extension of FOLTL, its model checking problem is similar to that of FOLTL. Recalling the validity relation for an automaton and an LTL formula we define validity for STAL formulas as follows.

Definition 6. $A \models \phi \iff \nexists w \in \prod(A)$ *such that* $A, w \models \neg\phi$

So STAL formulas are interpreted over infinite linear sequences of states (paths) and have linear counterexamples [20]. It follows that a counterexample of a specification $\phi \in \mathcal{S}$ is an infinite path $w \in \prod^{\infty}(A)$ such that $A, w \not\models \phi$ or $A, w \models \neg\phi$. One can show that w, as a counterexample to ϕ, can, w.l.o.g, be restricted to paths of the form $x.y^{\omega}$ [21]. Such a path is called a "*lasso*" and denoted by ℓ in this paper. A lasso consists of a finite prefix path x followed by an infinite loop over a finite suffix path y [20].

Lasso counterexamples are mainly counterexamples to *liveness properties* [22] which have a close relationship with infinite words over finite automata [23].

(i) *Node property keys are "*`Id`*", "*`Name`*", "*`Angle`*", "*`Speed_x`*"*$\_o_i$*, "*`Speed_y`*"*$\_o_i$*,* "`Distance_x`"$\_o_i$ and "`Distance_y`"$\_o_i$ for all $o_i \in Obj$.
(ii) *For all property keys k_i of $\sigma_i \in \Sigma$ pairs, edge property keys are "*`Id`*" and "*`input`*"*$\_k_i$.
(iii) *For all $q_i \in Q$ there exists $n_i \in N$ such that $P_{node}(n_i,$ "*`Name`*"$) =$ "q_i".*
(iv) *For all key-value pair (k_i, v_i) of $\sigma_i \in \Sigma, q_i$ and $q_{i+1} = \delta(\sigma_i, q_i)$ there exists $e_i \in E$ such* that $\lambda(e_i) = (n_i, n_{i+1})$ and $P_{edge}(e_i,$ "`input`"$\_k_i) = v_i$.
(v) *For all q_i and $o_i \in Obj$ then:*
-$P_{node}(n_i,$ "`Speed_x`"$\_o_i) = vel_x(o_i, q_i)$,
-$P_{node}(n_i,$ "`Speed_y`"$\_o_i) = vel_y(o_i, q_i)$,
-$P_{node}(n_i,$ "`Distance_x`"$\_o_i) = dist_x(o_i, q_i)$
-$P_{node}(n_i,$ "`Distance_y`"$\_o_i) = dist_y(o_i, q_i)$
- and $P_{node}(n_i,$ "`Angle`"$) = angle(q_i)$.

Fig. 8. Translation rules to convert an STA model A to its corresponding property graph $\mathcal{G}_A$.

For example, the formula $GF(\phi)$ specifies that the state property ϕ must hold infinitely often along an infinite path w. Clearly, a counterexample to this formula is an infinite path on which from some point on, ϕ does not hold. Intuitively, for a lasso counterexample $w = x.y^\omega$, this means ϕ never holds in the loop suffix, i.e., $A, y^\omega \not\models \phi$.

As with FOLTL, not all STAL counterexamples are infinite. Certain formulas have finite length counterexamples, i.e. satisfiability depends only on a finite prefix of a path. Examples are *safety properties* [22] which specify unsafe behavior that should never happen. An *invariant* is the simplest example of a safety property, i.e. a formula of the form $G(\phi)$, where ϕ has no modal operators. For invariants, a counterexample is a finite path where the last state violates ϕ. We can model check an STA A against a STAL formula $\phi \in \mathcal{S}$ as follows:

1) Translate A to a property graph $\mathcal{G}$.
2) Negate the requirement formula ϕ and translate this into a path pattern compatible with the target representation, e.g., a lasso pattern.
3) Execute a query to find matches for the pattern inside the property graph, i.e., a `MATCH` query in Cypher.
4) Return the results of pattern matching, if these exist, as paths, otherwise return *true*.

6.1 Soundness of Model Checking

The expressiveness of Cypher as a declarative query language is equivalent to a subset of first-order logic with transitive closure [18,24]. This enables Cypher to capture complex structural conditions and dependencies of STAL, and makes Neo4j a powerful platform for model checking. Thus we can translate a given STAL formula into a lasso graph query such that when evaluated over a graph representation of an STA model A, the query matches identify counterexamples.

Let $\mathcal{G}_A = (N, E, L, T, \lambda, Lab, Typ, P_{node}, P_{edge})$ be a graph representation of an STA $A = (\Sigma, Q, Obj, q_0, egoObj, \delta, angle, dist_x, dist_y, vel_x, vel_y)$ obtained by applying the rules in Fig. 8[5]. For any path $w \in \prod(A)$ in A we let $\overline{w} \in \prod(\mathcal{G}_A)$ denote the isomorphic copy[6] of w in the property graph $\mathcal{G}_A$.

Then Theorem 1 establishes the soundness of our model checking approach.

Theorem 1. *For any STAL formula ϕ there exists a Cypher query $\mu_\phi = \texttt{Trans}(\phi)$ such that for every lasso path $w = x.y^\omega \in \prod(A)$ $A, w \models \phi \Leftrightarrow \overline{w} \in [\![\mu_\phi]\!]_\mathcal{G}$.*

To prove Theorem 1, we first define the translation function $\texttt{Trans} : \mathcal{S} \rightarrow \mathcal{Q}$ that converts a STAL formula $\phi \in \mathcal{S}$ to a Cypher query μ_ϕ. Since our approach to STAL model checking is to search for lasso counterexamples, `Trans` coverts a given STAL formula into a *lasso query* which is composed of a lasso pattern

$$\pi_\ell = \texttt{(n\{Name:}``q_0"\texttt{\})-[e}_1\texttt{*0..]}\rightarrow\texttt{(m)-[e}_2\texttt{*1..]}\rightarrow\texttt{(m)}$$

and a `WHERE` condition, i.e.

$$\mu_\phi = \texttt{Trans}(\phi) = \texttt{MATCH}\ \pi_\ell\ \texttt{WHERE}\ \texttt{condition}(\overline{w}, \phi).$$

In the condition $\texttt{condition}(\overline{w},\phi)$ the path $\overline{w}$ is a generic solution to the structural lasso pattern π_ℓ that must be further filtered by the `WHERE` condition to satisfy the formula ϕ. Thus from Sect. 5.1, it follows that $[\![\mu_\phi]\!]_\mathcal{G} \subseteq [\![\pi_\ell]\!]_\mathcal{G}$. Since the Cypher structure of the lasso pattern is fixed, we need only define the Cypher expression $\texttt{condition}(\overline{w},\phi)$ inductively based on the structure of the STAL formula ϕ. The base case is where ϕ is atomic and does not include any modal operators.

Let $w = (\alpha_0, \alpha_1 ...)$ and $w^i = (\alpha_i, \alpha_{i+1} ...)$ where $\alpha_i = (\sigma_i, q_i)$ and $\sigma_i = (k_i, v_i)$ for $i \geq 0$. Notice that if w is a lasso then so is each w^i. Suppose $A, w \models \phi$ then we define `Trans` as follows:

Base Case. Since no modality is involved, the condition of ϕ must hold for the initial state $\alpha_0 \in w$, i.e., $A, (\mathbf{null}, q_0) \models \phi$. Similarly in $\overline{w} \in [\![\texttt{Trans}(\phi)]\!]_\mathcal{G}$, the condition of ϕ must hold for the initial state n_0 of $\overline{w}$. We define $\texttt{condition}(\overline{w}, \phi)$ for an atomic ϕ below.

$$A, w \models (exp_1 \mathbf{bop} exp_2) \iff [\![exp_1]\!]_{\alpha_0} \mathbf{bop}\ [\![exp_2]\!]_{\alpha_0} \equiv$$
$$\overline{w} \in [\![\mu_\phi = \texttt{MATCH}\ \pi_\ell\ \texttt{WHERE}\ \texttt{condition}(\overline{w}, (exp_1 \mathbf{bop} exp_2))]\!]_\mathcal{G}$$
$$\textit{where}\ \texttt{condition}(\overline{w}, (exp_1 \mathbf{bop} exp_2)) = (n_0.exp_1 \mathbf{bop}\ n_0.exp_2)$$

$$A, w \models (\texttt{input} = \sigma) \iff [\![\texttt{input} = \sigma]\!]_{\alpha_0} \equiv$$
$$\overline{w} \in [\![\mu_\phi = \texttt{MATCH}\ \pi_\ell\ \texttt{WHERE}\ \texttt{condition}(\overline{w}, (\texttt{input} = \sigma))]\!]_\mathcal{G}$$
$$\textit{where}\ \texttt{condition}(\overline{w}, (\texttt{input} = \sigma)) = (e_0.\texttt{input\_}k = v)$$

Inductive Case. For arbitrary formulas $\phi, \psi \in \mathcal{S}$ such that $w \models \phi$, $\overline{w} \in [\![\texttt{Trans}(\phi)]\!]_\mathcal{G}$, we define below `Trans` for $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $X(\phi)$, $F(\phi)$, $G(\phi)$ cases.

[5] In this figure, $n_i \in N$, $e_i \in E$, $L = \{$ "*State*"$\}$, $T = \{$"*Next*"$\}$, $Lab(n_i) =$ "*State*" and $Typ(e_i) =$ "*Next*".

[6] By the construction rules of Fig. 8, $\mathcal{G}_A$ is essentially structurally isomorphic to A.

- ***cases*** $(\neg\phi)$, $(\phi \wedge \psi)$ and $(\phi \vee \psi)$:

  $\texttt{Trans}(\neg\phi) \equiv \texttt{MATCH } \pi_\ell \texttt{ WHERE NOT condition}(\overline{w}, \phi)$
  $\texttt{Trans}(\phi \wedge \psi) \equiv \texttt{MATCH } \pi_\ell \texttt{ WHERE condition}(\overline{w}, \phi) \texttt{ AND condition}(\overline{w}, \psi)$
  $\texttt{Trans}(\phi \vee \psi) \equiv \texttt{MATCH } \pi_\ell \texttt{ WHERE condition}(\overline{w}, \phi) \texttt{ OR condition}(\overline{w}, \psi)$

- ***case*** $X(\phi)$: According to the semantics of STAL,

  $A, w \models X(\phi) \iff A, w^1 \models \phi \equiv$
  $\overline{w} \in [\![\mu_\phi = \texttt{MATCH } \pi_\ell \texttt{ WHERE condition}(\overline{w}^1, \phi)]\!]_\mathcal{G}$

- ***case*** $F(\phi)$: The semantic of the *eventually* operator F concerns a finite path from a state q_i to a reachable state q_j where ϕ holds. Therefore,

  $A, w \models F(\phi) \iff \exists j \in \mathbb{N} : A, w^j \models \phi \equiv$
  $\overline{w} \in [\![\mu_\phi = \texttt{MATCH } \pi_\ell \texttt{ WHERE Any}(n_i, e_i \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE condition}(\overline{w}^i, \phi))]\!]_\mathcal{G}$

  The `Any` function is a list predicate with boolean output which ensures that at least one element of a given list satisfies the conditions of its `WHERE` clause. Note that the index i of $\texttt{Any}(n_i, e_i \ldots)$ is a position index and the `Any` function is actually a loop that breaks when the `condition` is satisfied.

- ***case*** $A, w \models G(\phi)$: Evaluating an *always* operator G requires to verify ϕ on an infinite path and for a *lasso* path $w = x.y^\omega$, ϕ must be valid for all states and transitions of the lasso. Therefore all nodes of a lasso path $\overline{w}$ should satisfy the `WHERE` conditions of ϕ. In Cypher, the `All` function is a list predicate with boolean output which ensures that all elements of a given list satisfy the conditions of its `WHERE` clause. Note that the index i of $\texttt{All}(n_i, e_i \ldots)$ is a position index and the `All` function is actually a loop without any break condition.

  $A, w \models G(\phi) \iff \forall i \in \mathbb{N} : A, w^i \models \phi \equiv$
  $\overline{w} \in [\![\mu_\phi = \texttt{MATCH } \pi_\ell \texttt{ WHERE All}(n_i, e_i \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE condition}(\overline{w}^i, \phi))]\!]_\mathcal{G}$

Example 4. To clarify the translation procedure, below we provide two examples.

(i) $A, w \models GF(\phi) \rightarrow GF(\psi)$: This is a conjunction of GF and FG operators. If this formula is satisfiable by a lasso path $w = x.y^\omega$, then either all states of y must not satisfy ϕ, or at least one state of y must satisfy ψ.

$A, w \models GF(\phi) \rightarrow GF(\psi) \equiv FG(\neg\phi) \vee GF(\psi)$
$\iff (i, j \in \mathbb{N}, \exists i.\ \forall j, i \leq j.\ w^j \not\models \phi) \vee (i, j \in \mathbb{N}, \forall i.\ \exists j, i \leq j.\ w^j \models \psi) \equiv$
$\overline{w} \in [\![\mu_\phi = \texttt{MATCH } \pi_\ell \texttt{ WHERE}$
$\texttt{Any}(n_i, e_i \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE All}(n_j, e_j \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE NOT condition}(\overline{w}^{i+j}, \phi)))$
$\texttt{OR All}(n_i, e_i \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE Any}(n_j, e_j \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE condition}(\overline{w}^{i+j}, \phi)))]\!]_\mathcal{G}$

(ii) $A, w \models GFX(\phi)$: One of the complex structures is the combination of the *liveness* GF and the *next* X operators. However, the translation to a lasso query is straightforward.

$A, w \models GFX(\phi) \iff i, j \in \mathbb{N}, \forall i.\ \exists j, i \leq j.\ w^{j+1} \models \phi \equiv$
$\overline{w} \in [\![\mu_\phi = \texttt{MATCH } \pi_\ell \texttt{ WHERE All}(n_i, e_i \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE Any}(n_j, e_j \textit{ in } [\![\pi_\ell]\!]_\mathcal{G} \texttt{ WHERE}$
$\texttt{condition}(\overline{w}^{i+j+1}, \phi)))]\!]_\mathcal{G}$

Having defined the translation `Trans`, the proof of Theorem 1 is straightforward and relies on the definition of $\overline{w}$, $\overline{w} \in [\![\mu_\phi]\!]_{\mathcal{G}}$.

7 NeoMC Implementation and Evaluation

Figure 9 shows the architecture of our Neo4j-based model checker NeoMC that checks STAL formulas against STA models. As we have seen in Sect. 6, to check a STAL formula, NeoMC first converts an STA model to a Neo4j property graph. Then, it negates the formula and converts it into a Cypher pattern and uses this to perform a pattern matching query. If the query matches any paths in the graph, counterexamples are returned. Otherwise the verdict *true* is returned.

The Neo4j database (DB) is a stand-alone Java application that can be instantiated through the Neo4j API. It is responsible for performing all database queries and populating the results. The communication between NeoMC and the database is carried out over a TCP connection known as a "*Bolt url*".

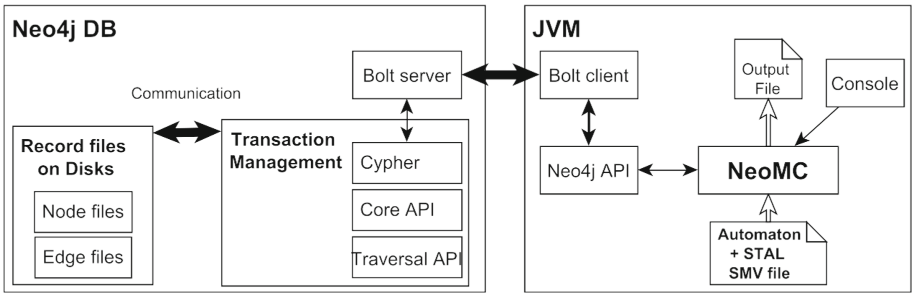

Fig. 9. Architecture of NeoMC integrated with Neo4j DB

To evaluate NeoMC we constructed a number of large STAs by machine learning using the platooning simulator of [25] to simulate a distributed multi-object dynamical system. These STAs ranged in size from 1 K to 71 K states. The largest STA had about 1.5 million transitions.

Our specific case study is a two vehicle platoon consisting of a leader (the ego object) and a follower. The leader is under manual control, and the follower is autonomously controlled using a cooperative adaptive cruise control (CACC) algorithm [25] for longitudinal control. The simulator accepts two input signals to control the brake and throttle of the lead vehicle. These continuous inputs were discretized to 10 different levels. In total, there were 21 discretized input values, called `Pedal` values. The outputs of the simulator used to construct the STA models were the speed of the leader and the relative distance between leader and the follower in the x dimension only, i.e., $\text{Speed}_x(\textit{leader})$ and $\text{Distance}_x(\textit{follower})$, denoted by Speed_x and Distance_x.

To benchmark NeoMC on the STAs derived from the platooning simulation, we defined a set of spatio-temporal requirements on the platooning vehicles themselves using STAL. Some of these requirements are presented in Table 1. A positive `Pedal` value in the table means pressing the lead vehicle gas pedal and a negative value means braking pedal level. Case (1) is to capture a near collision and means: "The distance between the follower and the lead vehicles should never be less than five meters. Case (2) means: "Gassing up the lead vehicle should eventually result in a speed greater than 30 km/h", and Case (3) means that, "Eventually the leader speed should stay at a high speed value greater than 70 km/h if the gas pedal is nearly fully pressed infinitely often".

Benchmarking tests of NeoMC were performed on an Ubuntu 16.4 LTS machine with Intel Core i5-6260U ×4 running at 1.80 GHz and 16Gb available RAM.

Table 2 summarizes our benchmark results for NeoMC model checking based on 10 different sized STA models and 7 different STAL requirements. Table 2 shows both the number of counterexamples found (lhs) and the execution time

Table 1. Platoon requirements in STAL.

Req	STAL formula ϕ	Cypher query $\mu = \texttt{Trans}(\neg\phi)$
(1)	$G\neg(\mathit{Distance}_x < 5)$	WHERE m.Distance_x < 5
(2)	$FG(\mathit{Pedal} > 0) \rightarrow FG(\mathit{Speed}_x > 30)$	WHERE Any(n_i, e_i in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE
		All(n_j, e_j in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE e_{i+j}.input_Pedal>0))
		AND All(n_i, e_i in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE Any(n_j, e_j in $[\![\pi_\ell]\!]_{\mathcal{G}}$
		WHERE NOT n_{i+j}.Speed_x>30))
(3)	$GF(\mathit{Pedal} > 7) \rightarrow GF(\mathit{Speed}_x > 70)$	WHERE All(n_i, e_i in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE
		Any(n_j, e_j in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE e_{i+j}.input_Pedal>7))
		AND Any(n_i, e_i in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE All(n_j, e_j in $[\![\pi_\ell]\!]_{\mathcal{G}}$
		WHERE NOT n_{i+j}.Speed_x>70))
(4)	$G(\mathit{Pedal} > 0 \rightarrow X(\mathit{acc}^* > 0))$	WHERE Any(n_i, e_i in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE e_i.input_Pedal>0
		AND NOT n_{i+1}.Speed_x - n_i.Speed_x>0)
(5)	$G(\mathit{Pedal} < 0 \rightarrow X(\mathit{acc} < 0))$	WHERE Any(n_i, e_i in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE e_i.input_Pedal<0
		AND NOT n_{i+1}.Speed_x - n_i.Speed_x<0)
(6)	$G\neg(\mathit{Pedal} = -10 \rightarrow$	WHERE Any(n_i, e_i in $[\![\pi_\ell]\!]_{\mathcal{G}}$ WHERE NOT e_i.input_Pedal=-10
	$(\mathit{Next}(\mathit{Speed}_x) - \mathit{Speed}_x) > 0)$	OR n_{i+1}.Speed_x - n_i.Speed_x < 20))
(7)	$G\neg(\mathit{Speed}_x > 120)$	WHERE m.Speed > 120

* The Leader acceleration.

Table 2. Number of identified counterexamples and the execution time for model checking of requirements in Table 1 for different model sizes (1.1 K–71 K states). Here ϵ means the execution time is less than 0.5 s.

Req	#Counterexamples on K-state models										Execution Time* (in seconds)									
	1.1	1.7	2.1	2.5	3.4	4	7.8	12.6	25	71	1.1	1.7	2.1	2.5	3.4	4	7.8	12.6	25	71
(1)	8	4	2	5	7	7	28	28	100	35	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	1	1	3	1
(2)	0	0	0	0	0	0	0	0	0	0	3	4	5	6	8	9	39	40	102	784
(3)	0	0	0	0	0	15	1	1	16	0	2	3	4	5	7	9	25	41	107	793
(4)	1	1	1	1	1	2	6	6	11	1	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	1	1
(5)	27	32	43	64	98	100	100	100	100	57	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	1
(6)	0	0	0	0	0	0	0	2	0	8	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	1	1
(7)	0	0	0	0	0	0	0	0	0	0	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	1	ϵ

in each case (rhs). We limited the maximum number of counterexamples to 100 to make the table concise and readable. In general, as the learned model grows in size, more violations of a requirement can be observed, because the model captures more execution paths with bad sequences of states. Table 2 shows that the execution time increases linearly with respect to model size.

We were unable to easily compare NeoMC performance with existing model checkers. One reason is that we could not find an efficient and scalable representation of large STA models for tools such as NuXMV [26], Spin [27] and LTSmin [28]. These tools parse the input models into their internal data representation and as the models grow in size, they either fail to read the files or construct the state space efficiently. Even for a medium size STA of 4k states, the model parsing times of Spin and the model checking time of NuXMV are beyond any acceptable figures. The memory usage of NuXMV is huge, of the order of tens of Gigabytes. Spin also consumes a lot of memory to generate its internal verifier. Only the built-in symbolic format (i.e. ETF) of LTSmin matches the STA models and quickly performs the model checking. However, the ETF format only works for symbolic datatypes and does not support FO STAL expressions and formulas.

8 Related Work

There is a large body of work on spatio-temporal logic. A rather complete list of related work in this area is provided in [1]. Verification of spatial and temporal modalities is studied in different domains such as in biochemistry [29], biology [30,31] and air traffic management [32]. Research on spatio-temporal model checking is often tailored to specific applications. SpaTeL [6] uses statistical model checking to estimate the probability of events in networked systems that relate different regions of space at different times. Statistical model checking has also been applied to collective adaptive systems where spatio-temporal properties expressed in STLCS [33] are verified against discrete, geographical models of a smart public transportation system [34]. In [7] a shape calculus based spatio-temporal model checking is introduced for the verification of a real-time railroad crossing system. A second order model checker is used to perform reachability checks on BDDs representing transition relations.

Verification of vehicle platooning is also studied by Kamali et al. [2] where timed and untimed automata models of a spatial controller are model checked using AJPF and UPPAAL. Schwammberger [3] introduced MLSL logic to verify safety of traffic maneuvers. Similarly to STAL, MLSL is using the snapshot concept which captures the state of objects at a given moment in time. However, our work differs from [2,3]. While they tried to verify safety of controller algorithms using timed-automata models in UPPAAL, our model checking technique is developed to verify a learned behavior of CO-CPSs using graph queries. Also, AJPF does not support temporal analysis and is resource-heavy, whereas graph databases scale with ease. The most closely related work to ours is [24] which used declarative graph queries for the verification of CPSs. They developed a runtime monitoring for railway systems against spatial requirements expressed in a 3-valued logic, but, this work lacks exhaustive verification and model checking.

9 Conclusions

We have proposed an approach to spatio-temporal model checking based on using the graph database Neo4j and its declarative query language Cypher. We have established the theoretical soundness of this approach, and implemented and evaluated it on a large case study. NeoMC shows that query solving for Cypher is an efficient way to implement model-checking. To the best of our knowledge, our work is the first attempt to apply graph database technology to model checking. Furthermore, Neo4j enabled us to quickly prototype a model checker for STAL that was scalable to large models. The efficiency of NeoMC is partly due to efficient search algorithms employed in modern graph databases, and also the fact that we could avoid constructing large product automata.

Acknowledgments. This research has been supported by KTH ICT-TNG project STaRT (Spatio-Temporal Planning at Runtime), as well as the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

1. Khosrowjerdi, H., Meinke, K.: Learning-based testing for autonomous systems using spatial and temporal requirements. In: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, MASES@ASE 2018, Montpellier, France, 3 September 2018, pp. 6–15 (2018). https://doi.org/10.1145/3243127.3243129
2. Kamali, M., Linker, S., Fisher, M.: Modular verification of vehicle platooning with respect to decisions, space and time. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2018. CCIS, vol. 1008, pp. 18–36. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-12988-0_2
3. Schwammberger, M.: An abstract model for proving safety of autonomous urban traffic. Theor. Comput. Sci. **744**, 143–169 (2018). https://doi.org/10.1016/j.tcs.2018.05.028
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
5. Chaochen, Z., Hoare, C., Ravn, A.P.: A calculus of durations. Inf. Process. Lett. **40**(5), 269–276 (1991). http://www.sciencedirect.com/science/article/pii/002001909190122X
6. Haghighi, I., Jones, A., Kong, Z., Bartocci, E., Grosu, R., Belta, C.: Spatel: a novel spatial-temporal logic and its applications to networked systems. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015, Seattle, WA, USA, 14–16 April 2015, pp. 189–198 (2015). https://doi.org/10.1145/2728606.2728633

7. Quesel, J.-D., Schäfer, A.: Spatio-temporal model checking for mobile real-time systems. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 347–361. Springer, Heidelberg (2006). https://doi.org/10.1007/11921240_24
8. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5), 68:1–68:40 (2017). http://doi.acm.org/10.1145/3104031
9. Bennaceur, A., Hähnle, R., Meinke, K. (eds.): Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8
10. Meinke, K., Niu, F.: A learning-based approach to unit testing of numerical software. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 221–235. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16573-3_16
11. Meinke, K., Sindhu, M.A.: Incremental learning-based testing for reactive systems. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 134–151. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21768-5_11
12. Webber, J.: A programmatic introduction to neo4j. In: Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH 2012, Tucson, AZ, USA, 21–25 October 2012, pp. 217–218 (2012). https://doi.org/10.1145/2384716.2384777
13. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, 10–15 June 2018, pp. 1433–1445 (2018). http://doi.acm.org/10.1145/3183713.3190657
14. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, (2010). iv + 417 pages, Machine Translation, vol. 24, no. 3–4, pp. 291–293, 2010. https://doi.org/10.1007/s10590-011-9086-9
15. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 11–139 (2008). https://doi.org/10.1145/1322432.1322433
16. Robinson, I., Webber, J., Eifrem, E.: Graph Databases: New Opportunities for Connected Data, 2nd edn. O'Reilly Media Inc., Sebastopol (2015)
17. Hölsch, J., Schmidt, T., Grossniklaus, M.: On the performance of analytical and pattern matching graph queries in neo4j and a relational database. In: Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, 21–24 March 2017 (2017). http://ceur-ws.org/Vol-1810/GraphQ_paper_01.pdf
18. Francis, N., et al.: Formal semantics of the language cypher. CoRR, vol. abs/1802.09984 (2018). http://arxiv.org/abs/1802.09984
19. Junghanns, M., Kießling, M., Averbuch, A., Petermann, A., Rahm, E.: Cypher-based graph pattern matching in gradoop. In: Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, 14–19 May 2017, pp. 3:1–3:8 (2017). http://doi.acm.org/10.1145/3078447.3078450
20. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
21. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths (extended abstract). In: 24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7–9 November 1983, pp. 185–194 (1983). https://doi.org/10.1109/SFCS.1983.51

22. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985). https://doi.org/10.1016/0020-0190(85)90056-0
23. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the Symposium on Logic in Computer Science (LICS 1986), Cambridge, Massachusetts, USA, June 16–18, 1986, pp. 332–344 (1986)
24. Búr, M., Szilágyi, G., Vörös, A., Varró, D.: Distributed graph queries for runtime monitoring of cyber-physical systems. In: Russo, A., Schürr, A. (eds.) FASE 2018. LNCS, vol. 10802, pp. 111–128. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_7
25. Meinke, K.: Learning-based testing of cyber-physical systems-of-systems: a platooning study. In: Reinecke, P., Di Marco, A. (eds.) EPEW 2017. LNCS, vol. 10497, pp. 135–151. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66583-2_9
26. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
27. Holzmann, G.J.: The SPIN Model Checker - Primer and Referencemanual. Addison-Wesley, Boston (2004)
28. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
29. Chiarugi, D., Falaschi, M., Hermith, D., Olarte, C.: Verification of spatial and temporal modalities in biochemical systems. Electr. Notes Theor. Comput. Sci. **316**, 29–44 (2015). https://doi.org/10.1016/j.entcs.2015.06.009
30. Parvu, O., Gilbert, D.R.: Automatic validation of computational models using pseudo-3D Spatio-temporal model checking. BMC Syst. Biol. **8**, 124 (2014). https://doi.org/10.1186/s12918-014-0124-0
31. Grosu, R., Smolka, S.A., Corradini, F., Wasilewska, A., Entcheva, E., Bartocci, E.: Learning and detecting emergent behavior in networks of cardiac myocytes. Commun. ACM **52**(3), 97–105 (2009). https://doi.org/10.1145/1467247.1467271
32. de Oliveira, Í.R., Cugnasca, P.S.: Checking safe trajectories of aircraft using hybrid automata. In: Anderson, S., Felici, M., Bologna, S. (eds.) SAFECOMP 2002. LNCS, vol. 2434, pp. 224–235. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45732-1_22
33. Ciancia, V., Grilletti, G., Latella, D., Loreti, M., Massink, M.: An experimental spatio-temporal model checker. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9509, pp. 297–311. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49224-6_24
34. Ciancia, V., Gilmore, S., Grilletti, G., Latella, D., Loreti, M., Massink, M.: Spatio-temporal model checking of vehicular movement in public transport systems. STTT **20**(3), 289–311 (2018). https://doi.org/10.1007/s10009-018-0483-8