International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Toolchain for Strategy Synthesis with Spatial Properties

Davide Basile · **Maurice H. ter Beek** · **Laura Bussi** ·
Vincenzo Ciancia

Received: date / Accepted: date

Abstract We present an application of strategy synthesis to enforce spatial properties. This is achieved by implementing a toolchain that enables the tools `CATLib` and `VoxLogicA` to interact in a fully automated way. The Contract Automata Library (`CATLib`) is aimed at both composition and strategy synthesis of games modelled in a dialect of finite state automata. The Voxel-based Logical Analyser (`VoxLogicA`) is a spatial model checker for the verification of properties expressed using the Spatial Logic of Closure Spaces on pixels of digital images. We provide examples of strategy synthesis on automata encoding motion of agents in spaces represented by images, as well as a proof-of-concept realistic example based on a case study from the railway domain. The strategies are synthesised with `CATLib`, while the properties to enforce are defined by means of spatial model checking of the images with `VoxLogicA`. The combination of spatial model checking with strategy synthesis provides a toolchain for checking and enforcing mobility properties in multi-agent systems in which location plays an important role, like in many collective adaptive systems. We discuss the toolchain's performance also considering several recent improvements.

Keywords Synthesis · games · spatial model checking · CAS · multi-agent systems · rigorous tool engineering

1 Introduction

Collective Adaptive Systems (CAS) consist of multiple spatially distributed agents, each exhibiting intricate autonomous behaviour. These agents possess the ability to cooperate towards shared objectives while typically also engaging in competition over limited resources [52].

Formal modelling and analysis are essential for enabling rigorous reasoning on the behaviour of CAS [59, 28]. By employing strategy synthesis, we facilitate the capacity of CAS to dynamically adapt their behaviour in accordance with evolving requirements. The seamless integration of strategy synthesis with well-established methods such as spatial model checking enhances the modelling and analysis techniques available for CAS.

Research on strategy synthesis in games is a hot topic, with established relations with supervisory control theory [70, 3], reactive systems synthesis [47], parity games [67] (with complexity breakthroughs [32]), automated behaviour composition [50], automated planning [33] and service coordination [16]. Several academic tools have been developed [45, 36, 38, 37, 62, 60, 69, 6] and applied to disparate domains, including land transport [14], maritime transport [72], medical systems [63], CAS [54], and autonomous agents path planning [55], in which problems are modelled as games and solved using tailored strategy synthesis algorithms.

In an automata-based setting, a strategy is a prescription of the behaviour (transitions) of a particular player for all possible situations (states) that leads that player to a specific goal (final state). Typically, there are other players or an environment with different, often competing goals to account for, and the set of transitions may be partitioned into controllable (by the particular player) and uncontrollable transitions. Strategy synthesis is concerned with the automatic computation of a (safe, optimal) strategy (controller) in such a game-based automata setting.

Recent advancements in spatial model checking have led to relevant results like the fully automated segmentation of regions of interest in medical images by brief,

Formal Methods and Tools Lab, ISTI–CNR, Pisa, Italy
E-mail: {basile,terbeek,bussi,ciancia}@isti.cnr.it

unambiguous specifications in spatial logic. The topological approach to spatial model checking of [42] is based on the Spatial Logic of Closure Spaces (SLCS). It provides a fully automated method to verify properties of points in graphs, digital images, and recently 3D meshes and geometric structures [29,66]. Spatial properties of points are related to topological aspects like being *near* to points satisfying a given property, or being able to *reach* a point satisfying a certain property, passing only through points obeying specific constraints.

The free and open-source tool `VoxLogicA` [27,39] has been designed from scratch for image analysis. Logical operators can be freely mixed with a few imaging operators, related to colour thresholds, texture analysis, or normalisation. The tool is quite fast, due to the following three factors: i) most primitives are implemented using the state-of-the-art imaging library SimpleITK[1]; ii) expressions are never recomputed (reduction of the syntax tree to a directed acyclic graph is used as a form of *memoisation*); iii) operations are implicitly parallelised on multi-core CPUs. Ongoing work (cf., e.g., [30]) is devoted to a GPU-based implementation which enables a speedup of 1–2 orders of magnitude.

Returning to the topic of strategy synthesis, the tool `CATLib` [6,18,5] is a library for performing compositions of contract automata [17] (a dialect of finite-state automata) and synthesising either their supervisory control, their orchestration, or their choreography [16], using novel notions of controllability [11]. `CATLib` offers scalability features such as a bounded on-the-fly state-space generation optimised with pruning of redundant transitions and parallel streams computations. Moreover, the software is free and open source [6], it has been developed using principles of model-based software engineering [5], and it has been extensively validated using various testing and analysis tools to increase confidence on the library's reliability.

Contribution. In this paper, we extend [10], where we proposed a new approach to combine strategy synthesis and spatial model checking. We proceed in a bottom-up fashion. First, we present a toolchain based on established off-the-shelf and tool-supported theories. We then explore the combination of `CATLib` and `VoxLogicA`, to concert the composition and synthesis functionalities of `CATLib` with the spatial model checking functionality of `VoxLogicA`. Subsequently, we provide proof-of-concept examples of strategy synthesis on automata encoding motion of agents in spaces represented by images[2]. The main insight is to encode an image as an automaton, whose states are the pixels of the image. These states are then interpreted as positions of an agent, and transitions to adjacent pixels represent motions of the agent. A composition of automata is thus a collective multi-agent system, in which each state of the composition is a snapshot of the current position of the agents in the map, and a game can thus be played by a set of agents against other opponent agents, where successful states and failure states can be identified using spatial model checking of the images. Consequently, the strategy is synthesised with `CATLib`, while the properties to enforce are defined by means of spatial model checking of the images with `VoxLogicA`. By updating the properties to enforce, it is possible to synthesise a new strategy enabling the agents to collectively adapt to the updated properties. The developed examples are open source and reproducible at [9].

This paper extends [10] into the following directions:

- the toolchain is automatically invoked by a main program (written in `python`) which takes care of receiving input parameters from the user, and automatically routing them appropriately when invoking the various phases of the toolchain;
- the number of images that are analysed by the spatial model checker has been drastically reduced;
- the performance of the toolchain has been improved by pruning unuseful computations and by reducing the size of the automata to be composed;
- the spatial model-checking procedure has been improved by resorting to a *batching* technique that permits to dramatically increase the computation-to-overhead ratio;
- an example from the railway domain [23] serves as a proof-of-concept of the practical applicability of the toolchain to real-world case studies, which also showcases the necessity of using the recent notion of *semi-controllability* rather than uncontrollability.

The benefits of our contribution can be summarised as follows: we show the practical applicability of the combination of two quite different tools, and we provide an original approach to strategy synthesis and spatial model checking, bridging theories and tools developed in different research areas.

Related Work. Practical application of spatial logics, including model checking, has been ongoing during the last decade. For instance, the research line originating in [57] merges spatial model checking with signal analysis. In the domain of cyber-physical systems, the approach of [76] demonstrates applications of SLCS in

[1] Cf. https://simpleitk.org/.

[2] In the `VoxLogicA` approach, images are seen as a special kind of graphs, where vertices are pixels, and edges represent proximity. Actually, the `VoxLogicA` family of tools can also operate on arbitrary directed graphs. Adapting the present work to the more general setting is left for future work.

a spatio-temporal domain with linear time, using bigraphical models. An abstract categorical definition of SLCS has been given in [35]. The spatial model checking approach of SLCS and `VoxLogicA` has been demonstrated in case studies ranging from smart transportation [41,40] and bike sharing [43,44] in the context of CAS, to brain tumour segmentation [4,27], labelling of white and grey matter [26], and contouring of nevi [25] in the context of medical imaging.

Synthesising strategies (or plans/control) for motion of agents is widely researched [65,55,56,2,73,48]. Spatial logics have been applied to this problem to investigate the synthesis of strategies from properties of spatially distributed systems specified with spatial logics [1,58,64]. Recently, the application domain of smart cities has been explored in [68], and the aforementioned signal-based approach has been enhanced for a hybrid approach to multi-agent control synthesis, by exploiting neural network and spatial-logical specifications in the Spatio-Temporal Reach and Escape Logic (STREL) formalism.

Differently from the above literature, we set out to integrate previously developed off-the-shelf algorithms and tools, with the aim of showing their applicability. Contract automata and their toolkit were introduced to synthesise ochestrations and choreographies of compositions of service contracts exchanging offers and requests [17,11,16,6]. The interpretation of an image as an (agent) contract automaton enables to connect contract automata and `CATLib` with spatial model checking and `VoxLogicA`, showing the flexibility of both approaches.

Structure of the Paper. After providing the necessary background on the tools `CATLib` and `VoxLogicA` in Section 2, we describe the proposed toolchain in Section 3, followed by a report on three experiments in Section 4, where we also discuss the improvement in performance with respect to [10]. Finally, we conclude and mention some future work in Section 5.

2 Background

In this section, we provide some background on the various formalisms and tools used in this paper.

2.1 `CATLib`, Automata Composition, and Strategy Synthesis

We first formally introduce contract automata and their synthesis operation. Contract automata are a dialect of Finite State Automata with a partitioned alphabet of actions. A Contract Automaton (CA) models either a single service or a multi-party composition of services performing actions. The number of services of a CA is called its *rank*. When *rank* = 1, the contract is called a *principal* (i.e., a single service). Figure 3 shows an example of a principal contract automaton. Labels of CA are vectors of atomic elements called *actions*. Actions are either *requests* (prefixed by `?`), *offers* (prefixed by `!`), or *idle* (denoted with a distinguished symbol `-`). Requests and offers belong to the (pairwise disjoint) sets R and O, respectively. The states of CA are vectors of atomic elements called basic states. Labels are restricted to be *requests*, *offers*, or *matches* where, respectively, there is either a single request action, a single offer action, or a single pair of request and offer actions that match, and all other actions are idle. The length of the vectors of states and labels is equal to the rank of the CA.

For example, the label `[!goright, ?goright]` is a match where the request action `?goright` is matched by the offer action `!goright`. Note the difference between a request label (e.g., `[?goright, -]`) and a request action (e.g., `?goright`). A transition may also be called a request, offer, or match according to its label.

The goal of each service is to reach an accepting (*final*) state such that all its request (and possibly offer) actions are matched. In [11], CA were equipped with *modalities*, i.e., *necessary* ($\Box$) and *permitted* ($\Diamond$) transitions, respectively. Permitted transitions are controllable, whereas necessary transitions can be uncontrollable or semi-controllable. The resulting formalism is called *Modal Service Contract Automata* (MSCA). In the following definition, given a vector $\vec{a}$, its ith element is denoted by $\vec{a}_{(i)}$.

Definition 1 (MSCA) Given a finite set of states $\mathcal{Q} = \{q_1, q_2, \ldots\}$, an MSCA $\mathcal{A}$ of rank n is a tuple $\langle Q, \vec{q}_0, A^r, A^o, T, F\rangle$, with set of states $Q = Q_1 \times \ldots \times Q_n \subseteq \mathcal{Q}^n$, initial state $\vec{q}_0 \in Q$, set of requests $A^r \subseteq \mathsf{R}$, set of offers $A^o \subseteq \mathsf{O}$, set of final states $F \subseteq Q$, set of transitions $T \subseteq Q \times A \times Q$, where $A \subseteq (A^r \cup A^o \cup \{\bullet\})^n$, partitioned into *permitted* transitions $T^{\Diamond}$ and *necessary* transitions $T^{\Box}$, such that: (i) given $t = (\vec{q}, \vec{a}, \vec{q}\,') \in T$, $\vec{a}$ is either a request, an offer, or a match; and (ii) $\forall i \in 1 \ldots n$, $\vec{a}_{(i)} = \bullet$ implies $\vec{q}_{(i)} = \vec{q}\,'_{(i)}$.

Composition of services is rendered through the composition of their MSCA models by means of the *composition operator* $\otimes$, which is a variant of a synchronous product. This operator basically interleaves or matches the transitions of the component MSCA, but, whenever two component MSCA are enabled to execute their respective request/offer action, then the match is forced to happen. Moreover, a match involving a necessary transition of an operand is itself necessary. The rank of the composed MSCA is the sum of the ranks of its operands. The vectors of states and actions of the composed MSCA are built from the vectors of states and actions of the component MSCA, respectively.

In a composition of MSCA, typically various properties are analysed. We are especially interested in *agreement*. The property of agreement requires to match all requests, whilst offers can go unmatched.

CA support the synthesis of the most permissive controller (mpc) from the theory of supervisory control of discrete event systems [70,34], where a finite state automaton model of a *supervisory controller* (called a strategy in this paper) is synthesised from given (component) finite state automata that are composed. Supervisory control theory has been applied in a variety of domains [53,74,71,24,49,16,75], including healthcare. In this paper, we use the synthesis in the framework of games, whose relation with supervisory control is well known [3].

The synthesised automaton, if successfully generated, is such that it is *non-blocking*, *controllable*, and *maximally permissive*. An automaton is said to be *non-blocking* if from each state at least one of the *final states* (distinguished stable states that represent completed 'tasks' [70]) can be reached without passing through so-called *forbidden states*, meaning that the system always has the possibility to return to an accepted stable state (e.g., a final state). The algorithm assumes that final states and forbidden states are indicated for each component. The synthesised automaton is said to be *controllable* when only controllable actions are disabled. Indeed, the supervisory controller is not permitted to directly block uncontrollable actions from occurring; the controller is only allowed to disable them by preventing controllable actions from occurring. Finally, the fact that the resulting supervisory controller is said to be *maximally permissive* (or least restrictive) means that as much behaviour of the uncontrolled system as possible is still present in the controlled system without violating neither the requirements, nor controllability, nor the non-blocking condition.

Finally, we recall the specification of the abstract synthesis algorithm of CA from [16]. This algorithm will be used to synthesise a strategy for the spatial game in the next sections. The synthesis of a controller, an orchestration, and a choreography of CA are all different special cases of this abstract synthesis algorithm, formalised in [16] and implemented in `CATLib` [5] using map reduce style parallel operations of Java Streams. This algorithm is a fix-point computation where at each iteration the set of transitions of the automaton is refined (pruning predicate ϕ_p) and a set of forbidden states R is computed (forbidden predicate ϕ_f). The synthesis is parametric on these two predicates, which provide information on when a transition has to be pruned from the synthesised automaton or a state has to be deemed forbidden. We refer to MSCA as the set of (MS)CA, where the set of states is denoted by Q and the set of transitions by T (with $T^{\square}$ denoting the set of necessary transitions). For an automaton $\mathcal{A}$, the predicate $\mathit{Dangling}(\mathcal{A})$ contains those states that are not reachable from the initial state or that cannot reach any final state. Let $\mathbb{B} = \{\top, \bot\}$ denote the Boolean constants true ($\top$) and false ($\bot$).

Definition 2 (abstract synthesis [16]) Let $\mathcal{A}$ be an MSCA, $\mathcal{K}_0 = \mathcal{A}$, and $R_0 = \mathit{Dangling}(\mathcal{K}_0)$. Given two predicates $\phi_p, \phi_f : T \times \mathit{MSCA} \times Q \to \mathbb{B}$, let the *abstract synthesis function* $f_{(\phi_p,\phi_f)} : \mathit{MSCA} \times 2^Q \to \mathit{MSCA} \times 2^Q$ be defined as follows:

$$f_{(\phi_p,\phi_f)}(\mathcal{K}_{i-1}, R_{i-1}) = (\mathcal{K}_i, R_i), \text{ with}$$

$$\begin{aligned} T_{\mathcal{K}_i} &= T_{\mathcal{K}_{i-1}} - \{\, t \in T_{\mathcal{K}_{i-1}} \mid \phi_p(t, \mathcal{K}_{i-1}, R_{i-1}) = \top \,\} \\ R_i &= R_{i-1} \cup \{\, \vec{q} \mid (\vec{q} \rightarrow) = t \in T^{\square}_{\mathcal{A}}, \phi_f(t, \mathcal{K}_{i-1}, R_{i-1}) = \top \,\} \\ &\quad \cup \mathit{Dangling}(\mathcal{K}_i) \end{aligned}$$

The abstract controller is defined in Equation 1 below as the least fixed point (cf. [16, Theorem 5.2]) where, if the initial state belongs to $R_s^{(\phi_p,\phi_f)}$, then the controller is empty; otherwise, it is the automaton with the set of transitions $T_{\mathcal{K}_s^{(\phi_p,\phi_f)}}$ and without states in $R_s^{(\phi_p,\phi_f)}$.

$$(\mathcal{K}_s^{(\phi_p,\phi_f)}, R_s^{(\phi_p,\phi_f)}) = \mathit{sup}(\{\, f^n_{(\phi_p,\phi_f)}(\mathcal{K}_0, R_0) \mid n \in \mathbb{N} \,\}) \quad (1)$$

***CATLib*.** Contract automata and their functionalities are implemented in a software artefact, called Contract Automata Library (`CATLib`), which is under continuous development [6]. This software artefact is a by-product of scientific research on behavioural contracts and implements results that have previously been formally specified in several publications (cf., e.g., [18,19,17,22,21,12,20,13,16,11]). `CATLib` has been designed to be easily extendable to support similar automata-based formalisms. Currently, `CATLib` also supports synchronous communicating machines [61,46]. `CATLib` and the other CA tools [7] allow programmers to use CA for developing more reliable applications. In this paper, we further showcase the flexibility of `CATLib` by using it to synthesise strategies for mobile agents in spatial games. `CATLib` has been implemented using modern established technologies for building, testing, documenting, and delivering high quality source code. `CATLib` is tested up to 100% coverage of all lines, branches, and the strength of the tests is measured with mutation testing with top score.

2.2 VoxLogicA, Spatial Model Checking, and Image Analysis

The *Spatial Logic of Closure Spaces* (SLCS) is a modal logic language equipped with a unary 'nearness' modality and two binary operators: 'reaches' and 'is reached'. The language is interpreted on points of a spatial structure, which is, generally speaking, a *Closure Space* (cf. [42] for details). Graphs, digital images, topological spaces, and simplicial complexes are all instances of closure spaces.

In this paper, we concentrate on the interpretation of SLCS on images. In this case, the two reachability modalities collapse and the nearness modality is a derived operator based on the reachability operator, causing a particularly simple syntax.

Definition 3 Fix a set AP of atomic propositions. The syntax of SLCS is defined by the following grammar:

$$\phi ::= p \mid \top \mid \neg\,\phi \mid \phi \wedge \phi \mid \rho\,\phi[\phi]$$

where $p \in AP$.

Models of SLCS formulae, for the purpose of this paper, are the pixels of digital images; i.e., each SLCS formula induces a truth value for each point of a given digital image. In order to define the interpretation of formulae, a notion of *path* needs to be established, based on a notion of *neighbourhood* or *connectivity* of pixels. VoxLogicA uses the so-called '8-neighbourhood', i.e., each pixel is adjacent to 8 other pixels, namely those that share an edge or a vertex with it. Adding a notion of connectivity permits one to interpret the set of pixels of an image as a (symmetric) graph. Graph-theoretical paths are then well defined, and used below.

The interpretation of formulae depends upon a valuation of atomic propositions, assigning to each atomic proposition the set of points on which it holds, and assigning a direct interpretation to the symbols $p \in AP$. The meaning of the truth value $\top$ (true), negation ($\neg$), and conjunction ($\wedge$) is the usual one. A pixel x satisfies $\rho\,\phi_1[\phi_2]$ if there is a path rooted in x, reaching a pixel satisfying ϕ_1, such that all intermediate points, except eventually the extremes, must satisfy ϕ_2. We make use of the derived operator $\phi_1 \rightsquigarrow \phi_2$ which is similar to $\rho\,\phi_2[\phi_1]$, but the extremes are also required to satisfy ϕ_1. The *near* derived operator $\mathcal{N}\phi \triangleq \rho\,\phi[\neg\top]$ is true at point x if and only if there is a pixel adjacent to x where ϕ holds.

From now on, we use the tool's syntax, which uses `tt`, `&`, `|`, `!`, `~>`, and `N` for $\top$, conjunction, disjunction, negation, $\rightsquigarrow$, and $\mathcal{N}$, respectively, permits macro abbreviations of the form `let identifier = expression`, permits function definitions of the form `let identifier(argument1,...,argumentN) = expression`, and it also permits other constructs not needed for the scope of this paper. On images, atomic propositions can be expressions predicating over the colour components of the pixels. For instance, in our example specification (cf. Figure 7), to characterise the pixels composing a door as the blue pixels (note that 255 is the maximum value since we are using 8-bit images), given that `img` denotes an image, we use:

```
let r = red(img)
let g = green(img)
let b = blue(img)
...
let door = (r =. 0) & (b =. 255) & (g =. 0)
```

Also, the tool permits global formulae that assign a truth value to models, not just pixels in isolation. These can be based on the `volume(phi)` primitive, that computes the number of pixels satisfying the formula `phi`. For instance, existential and universal quantification are defined as follows:

```
let exists(p) = volume(p) .>. 0
let forall(p) = volume(p) .=. volume(tt)
```

We note that the type system of VoxLogicA is very simple, and comprises *numbers*, *Boolean values*, *images of numbers* (single-channel images, sometimes called *grayscale*), *images of Boolean values*, very often called *binary images* or *masks*, and ordinary *multi-channel images*. Operators are strongly typed with no type overloading. Therefore, for instance, the pixel-by-pixel *and* of two Boolean-valued images is a different operator with respect to the conjunction of two Boolean values, and it also differs from the conjunction of the Boolean value of each pixel of an image with a Boolean (scalar) constant. With some exceptions, the naming convention of operators reflects their type, having a dot on the side of the 'scalar' value (Boolean or number) and no dot on the side of the image, so for instance `.&.` is Boolean *and*, whereas `&` is pixel-by-pixel *and* of two images. With respect to Figure 7, for instance, we have that `base` and `img` are multi-channel images, with the operators `red`, `green`, `blue`, extracting number-valued images from them. The definition of `mrRed` (a red area) contains the `=.` operator taking a number-valued image on the left, and a number on the right (hence the dot on the right side). In the definition of the property `forbidden1`, one can find an example of the use of the operator `.|.` which takes as arguments two Boolean values.

3 Tool Methodology

In this section, we discuss the tool methodology used to chain CATLib and VoxLogicA in order to perform strategy synthesis of spatial properties. The diagram in Figure 1 depicts the workflow and the various activities in which the whole process is decomposed.

The process starts with a PNG image, depicting a map or planimetry, for agents to move in. Note that this implies that the state space is discrete, finite, and can be provided by a user with no training on the underlying theories used. Further input concerns the spatial properties that one wants to enforce with the synthesised strategy, modelling the forbidden configurations to avoid and the final configurations to reach, as well as the number of agents in the experiments with their starting position, and an indication of which agents are the controllable players and which are the uncontrollable opponents. The aim of the process is to produce the *maximally permissive* strategy for moving the players against all possible moves of the opponents, such that no forbidden configuration is ever reached and it is always possible to reach a final configuration. In game-theoretical jargon, this is both a safety game and a reachability game [15]. The strategy is maximal, in the sense that it includes all possible behaviour that satisfies the above properties. If the strategy is empty, then there exists no strategy for the players satisfying the given properties. CATLib only considers finite traces: infinite looping behaviour where an agent is stalled and is prevented from reaching a reachable final configuration is ruled out.

CATLib Activities. To allow the integration of CATLib and VoxLogicA, CATLib has been extended to allow the import/export of PNG images, which are internally converted into automata. These automata have pixels as states and transitions connecting adjacent pixels. We interpret these automata as agents, whose position is represented by the current state and transitions are requests to move up, down, left, or right to adjacent pixels/states. Note that more transitions could be added to model complex movements (e.g., between pixels that are not adjacent) should this be required by the specific application. The experiments in this paper did not require such movements. If a border is reached, then there will be no request transition in the automaton to move beyond that border. Each state is labelled with both a position, rendered in three coordinates (the third coordinate is currently not used), and the colour of the pixel. Figure 2 depicts a small portion of an agent automaton.

A driver automaton is used to command an agent to move in a specific direction. It is depicted in Figure 3. The driver can impose some constraints (e.g., never go down). The last automaton that is used models a door, which is initially closed, and which can be opened and closed repeatedly. It is depicted in Figure 4.

The first activity of CATLib thus consists of importing and creating such automata. There can be several instances of agents and doors or different maps according to the parameters of the experiments to perform.

The second activity consists of composing these automata to generate all possible reachable and valid configurations. As stated in Section 2, the composition has unicast synchronisations between offers and requests of agents (called matches), and labels that are only single moves of an agent performing an offer. Agents who perform requests can move only when paired with a corresponding offer. This type of synchronised behaviour is called agreement: all requests must be matched.

Note that this is an optimisation with respect to the previous version of the toolchain as discussed in [10]. Indeed, in [10], the composition included all possible behaviours, including invalid ones. We refined this activity to avoid the generation of invalid states in the composition. Previously, this was done in an additional activity performed later on, which has been removed. In Section 4, we will discuss the improvement in performance.

To reduce the size of the state space, the composition of CATLib allows to avoid generating portions of the state space that are known to violate some property. These are, e.g., configurations where an agent is placed on top of a wall (i.e., its state has colour #000000), on top of another agent (i.e., in a state of the composition, two agents have the same coordinates), or on top of a closed door (i.e., in a state of the composition, one agent has the same coordinates as the door and the door is closed). Since these are simple invariant properties (it only suffices to check the labels of states), they can be directly checked in CATLib. VoxLogicA is used to evaluate more complex spatial properties (cf. Figure 7 below). The aforementioned invalid moves are also specified in VoxLogicA under the property wrong in Figure 7 below.

In case of controllable 'bad' transitions, these will not be generated since they will be pruned by the synthesis. In case of uncontrollable 'bad' transitions, these will be generated (since they cannot be pruned) but their target state will not be visited (the synthesis will try to make these 'bad' states unreachable). Thus, once some agent is rendered as not controllable (by changing its transitions), it cannot be stopped from reaching an illegal configuration. It follows that illegal configu-

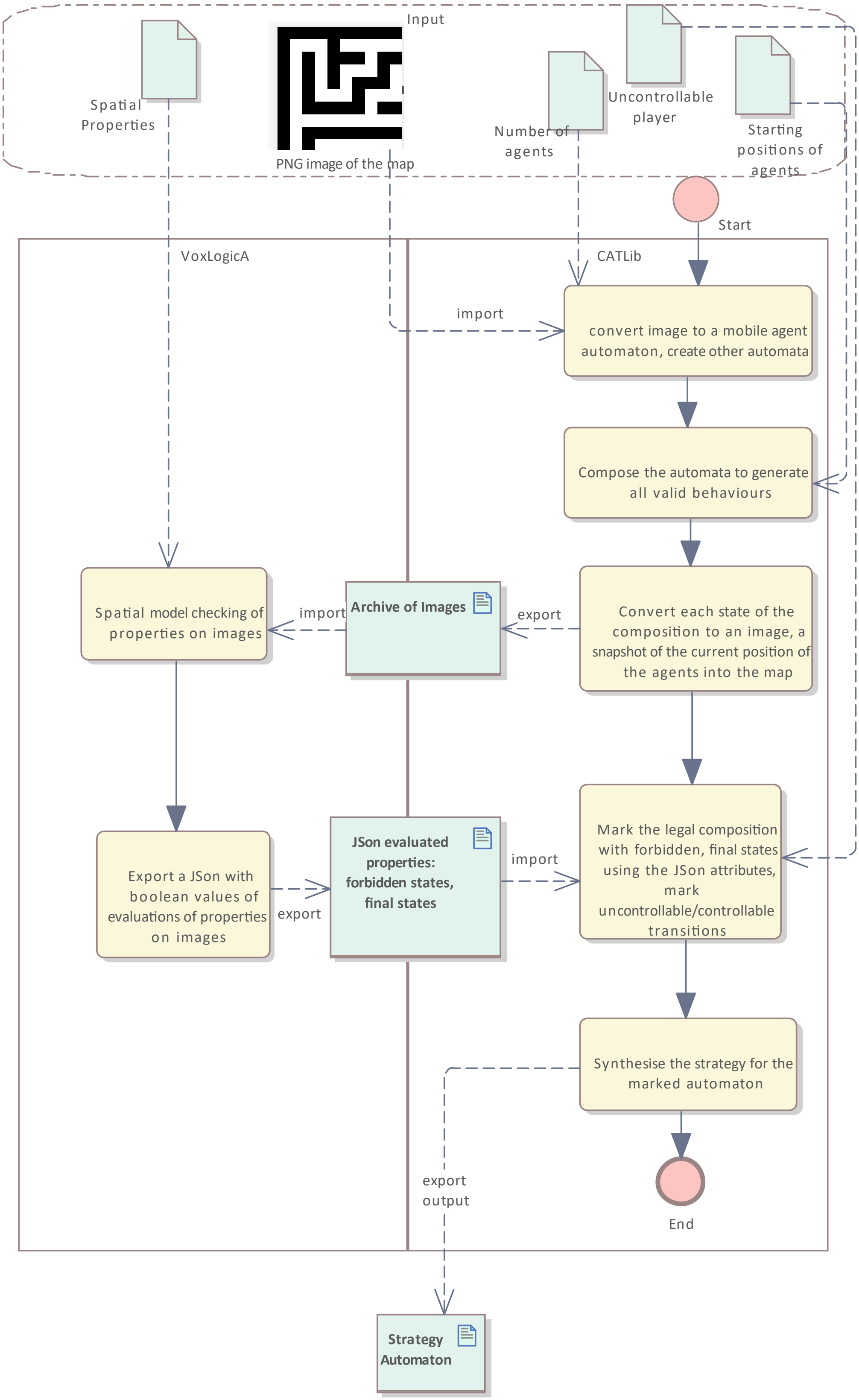
Input
Spatial Properties
PNG image of the map
Number of agents
Uncontrollable player
Starting positions of agents
Start
VoxLogicA
CATLib
import
convert image to a mobile agent automaton, create other automata
Compose the automata to generate all valid behaviours
Spatial model checking of properties on images
import
Archive of Images
export
Convert each state of the composition to an image, a snapshot of the current position of the agents into the map
Export a JSon with boolean values of evaluations of properties on images
export
JSon evaluated properties: forbidden states, final states
import
Mark the legal composition with forbidden, final states using the JSon attributes, mark uncontrollable/controllable transitions
Synthesise the strategy for the marked automaton
End
export output
Strategy Automaton

Fig. 1 The workflow showing the integration of the two tools

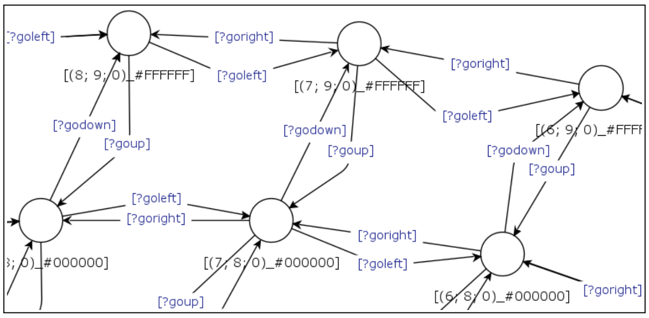

Fig. 2 A zoom-in on a fragment of the agent automaton

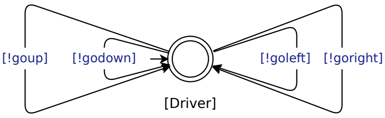

Fig. 3 The driver automaton

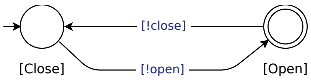

Fig. 4 The door automaton

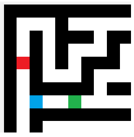

Fig. 5 State `[(10; 10; 0)_#FFFFFF, (5; 7; 0)_#FFFFFF, Driver, Close]` of the composition of two agents, a driver, and a door. The door is in position `(2; 7)` and is closed. The first agent is depicted red, the second is green, and the door is blue. The attributes of the position of the two agents are both `#FFFFFF`, which is the hexadecimal value for the colour white, i.e., both agents are placed on a white cell of the map.

rations must be removed before deciding which agents are not controllable and which are controllable. In this step it is also decided what are the initial positions of the agents, i.e., the initial state where the state-space generation starts. Indeed, the generated state space also depends on the given initial conditions.

In the composed automaton, each state is a tuple of states of all agents (including the door and the driver). Each state can be represented as an image, a snapshot of the current configuration. For instance, Figure 5 depicts a state rendered as an image. The image is generated by colouring the starting `PNG` image with a red, green, and blue pixel to indicate where, respectively, the first agent, the second agent, and the door are located. The door is only coloured when it is closed.

The third activity consists of generating all images for all states of the composition. These images are then passed to `VoxLogicA` (whose activities are described below) to evaluate for all properties whether or not they are satisfied. The number of images generated at this step are far fewer than those that were generated in [10], because the composed automaton has been polished by removing invalid states.

Next, the composed automaton must be marked with those states that are forbidden and those that are final. This is the fourth activity of `CATLib`. Also, it must be decided which agents are controllable and which not. This information is provided partly as input parameters of the experiments, and partly as a `JSon` file computed by `VoxLogicA`, where each state has as set of Boolean attributes, one for each evaluated spatial property.

After all states and transitions have been marked with the required information, the strategy synthesis is performed as the final, fifth activity of `CATLib`. The algorithm computes the maximal behaviour of the composition (in agreement) such that it is always possible to reach a final configuration and forbidden configurations are never traversed. If the strategy is non-empty, this will provide information on the behaviour to be followed by the controllable agents to ensure that a final configuration is always reached without passing through forbidden configurations, against all possible moves of uncontrollable components.

`VoxLogicA` Activities. The first activity of `VoxLogicA` is the evaluation of the formulae representing final and forbidden states. This is done via an auxiliary `python` script, that takes as input the logical specification, written into a file with `.imgql` extension, the base image (i.e., the map or planimetry where agents move), and the directory containing all reachable configurations, encoded as images. The `python` script then iterates the specification on the whole dataset of input images.

The second activity of `VoxLogicA` collects all the properties that have been computed in the first activity, locally for each state, and turns them into a single source of information, in the form of a `JSon` file that contains a record for each state, reporting on all the properties that have been described in the specification. In order to do so, a special output mode of `VoxLogicA` is used, where the tool outputs a single `JSon` record of all the user-specified properties that have been printed or saved in the specification.

Parameterisation. Previously, in [10], the parameters of the experiments were hardcoded and each different setup required to update the source code. We updated the toolchain such that the parameters are now passed to the toolchain at command line. This improves the usability of the toolchain and allows its automatisation. Below we only report the most significant parameters:

`experiment [1|2|3]` This is a special option that allows to select the setup of either experiment 1, 2, or 3 (cf. Section 4). If this option is not specified, one has to provide the setup with the parameters below;

`gateCoordinates x y` This parameter is used to set the coordinates of the gate;

`position_agent_1 x y` This parameter is used to set the initial coordinates of the first agent;

`position_agent_2 x y` This parameter is used to set the initial coordinates of the second agent;

`controllability [1|2]` If the value is 1 (2, respectively), then the agents (gate, respectively) are controllable, but not the gate (agents, respectively).

`specification` This parameter indicates the `ImgQL` specification to be used as input of `VoxLogicA`.

Other parameters include the various paths of the files to load/store, the name of the attributes in the output provided by `VoxLogicA`, and the selection of an activity to execute (e.g., image generation and strategy synthesis).

Handling Higher Dimensions. Although in this paper, we only develop examples using 2-dimensional images, the toolchain enables the representation of scenarios with higher dimensions. An n-dimensional object can be represented in `CATLib` as an automaton whose state is a vector of coordinates of length n (currently, three dimensions are supported). On the logical side, we remark that the main case study for `VoxLogicA` uses medical 3D images [27]. Furthermore, the model checker has been ported to the analysis of 3D meshes in [29]. Finally, the logic SLCS has also been interpreted on higher-dimensional simplicial complexes, which are capable of expressing logical relationships between data, as demonstrated in [66]. However, it must be noted that higher dimensions may render analysis unfeasible unless specific computational methods are used.

User Input. The way in which a user can provide the images in input to the toolchain is not constrained, and it depends on the specific problem. The automata used by `CATLib` are generated from the input image. Later, the images generated by `CATLib` are interpreted as graphs for model-checking purposes by `VoxLogicA`. Note that the input image provided by the user is different from the images that are generated by `CATLib` and model checked by `VoxLogicA`. Indeed, the graph representation used by `VoxLogicA` is not related to the behavioural graph (i.e., the contract automaton) used by `CATLib`, but it only depends on an image's structure.

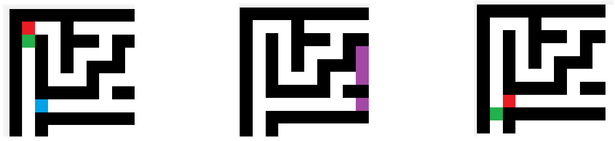

Fig. 6 On the left, the initial configuration of both experiments. In the middle, the final states of the first experiment (marked in violet). On the right, a configuration traversed by one of the shortest paths of the first experiment's strategy, in which the red agent is crossing the door before the green agent does, thus avoiding forbidden configurations.

4 Experiments

In this section, we describe the experiments that have been performed following the process described in the previous section. We performed three experiments.

The first two experiments, discussed in Section 4.1, start from the same initial conditions, but with opposite controllable/uncontrollable agents and forbidden/final states. The setup and outcome of these two experiments are reported in Table 1. The third experiment, discussed in Section 4.2, is based on a railway scenario. The setup and outcome of this experiment are reported in Table 2.

The repository, publicly available [9], contains all data, sources, and step-by-step instructions on how to use the toolchain to reproduce the experiments.

4.1 Maze Examples

The `PNG` map image used as planimetry is a 10 × 10 pixels image that weighs 188 bytes. It is depicted in Figures 1 and 5 (without coloured pixels). This image was generated using one of the many maze generators available through the Internet.

The setup for the experiments is that of two duplicate mobile agents, one door agent, and one driver agent. The door agent is placed in position `(2; 7)` (cf. Figure 5). Initially, the red agent is in the top left corner of the white corridor (position `(1; 1)`), whereas the green agent is just below the red one (position `(2; 1)`) and the door is closed. The initial state is depicted in Figure 6 (left).

The illegal moves were described in the previous section. We recall that in a legal composition, no agent moves over a wall, a closed door, or another agent.

Below we report the invocation of the composition function of `CATLib`. The composition is instantiated with the list of operands, namely the two agents, the driver, and the door. The second argument is the pruning predicate: if a generated transition satisfies the pruning predicate it will be pruned and not further explored. When applying the composition it is possible to specify a bound on the maximum depth of the generated

Table 1 Summary of the two experiments

	First experiment	**Second experiment**
Controllable	Red and green agents	Door
Uncontrollable	Door	Red and green agents
Initial state	Green agent in front of red agent	Green agent in front of red agent
Final states	Both the red and the green agent reached the exit	The door separates the green agent on the right from the red agent on the left
Forbidden states	The door separates the green agent on the right from the red agent on the left, or the red and green agents are not near each other	Both the red and the green agent reached the exit
Strategy	The red and green agents switch position before traversing the door	Empty

Table 2 Summary of the third experiment

	Third experiment
Semi-controllable	Trains
Controllable	Semaphore
Initial state	Both trains behind the semaphore
Final states	Both trains outside the junction area
Forbidden states	Both trains inside the junction area or one train is inside the junction area, while the other train is before the semaphore and the semaphore is open
Strategy	The semaphore is opened for letting the first train enter the junction area. After it does, the semaphore is closed. Once the first train exits the junction area, the semaphore opens again to let the second train enter the junction area so that both trains can eventually exit the junction area

automaton. In this case, the bound is set to the maximum Integer value. The two agents are instantiated with `maze_tr` and `maze2_tr` being their set of transitions, which only differ in the initial state. The property of agreement is passed as a lambda expression: transitions with a request label will be pruned. Similarly, this condition is put in disjunction with a condition checking whether the target state of the generated transition is 'bad' (i.e., an illegal transition), in which case the transition is pruned.

```
MSCACompositionFunction<String> cf =
 new MSCACompositionFunction<>(List.of(new Automaton<>(maze_tr),
  new Automaton<>(maze2_tr),driver,door),
   t->t.getLabel().isRequest() || badState.test(t.getTarget()));
Automaton<String, Action, State<String>,
 ModalTransition<String,Action,State<String>,CALabel>>
  comp = cf.apply(Integer.MAX_VALUE);
```

Spatial Properties. In the first experiment, the final and forbidden states are set according to the following definitions. Consider the specification given in Figure 7 (cf. Section 2 for an introduction to the operators used therein). The final state is set to be that on the right-hand side of the image passing through the corridor where the door is located (property `final1`), and is depicted in Figure 6 (middle). The property `forbidden1` identifies forbidden states as the disjunction of three sub-properties, characterising: 1) illegal states (property `wrong`); 2) states in which the two agents are in two areas separated by the closed door, and the green agent is on the right side of the door, i.e., it can reach an escape (a final state), whereas the red agent cannot because it is blocked by the door (property `greenFlees`); and 3) states in which the agents are not close to each other (`!. nearby`). In fact, Figure 5 represents one of these forbidden states.

In the second experiment, we change the setting according to what is summarised in Table 1. The final state becomes an illegal state of the first experiment, such that we end up with an empty strategy. Hence, the property `final2` is the disjunction of two previously forbidden properties: `greenFlees` (i.e., the green agent can reach an exit, while the red one cannot) and `wrong`, namely, an illegal situation that might consist of agents over the door, agents over a wall, or overlapping agents. Conversely, property `forbidden2` is set to be the final state of experiment 1, so we simply get `forbidden2 = final1` (cf. Figure 7). Thus the forbidden state is now the one in which both agents reach the exit.

Synthesis. Finally, in this first experiment we interpret the door as uncontrollable, whereas the red and green agents are controllable. Basically, this is a scenario in which the two players are playing against an uncontrollable door. Below we list the code used to invoke the synthesis operation of `CATLib`. The instantiation of the operation takes as argument the property to enforce, agreement in this case, and the automaton where the synthesis is applied, called `marked` in this case.

```
Automaton<String, Action, State<String>,
 ModalTransition<String,Action,State<String>,CALabel>>
  strategy = new MpcSynthesisOperator<String>(new Agreement()).
   apply(marked);
```

```
let r = red(img)
let g = green(img)
let b = blue(img)
let rb = red(base)
let gb = green(base)
let bb = blue(base)

let exists(p) = volume(p) .>. 0
let forall(p) = volume(p) .=. volume(tt)
let forallin(x,p) = forall( (!x) | p)

let door = (r =. 0) & (b =. 255)
    & (g =. 0)
let floorNoDoor = (rb =. 255)
    & (bb =. 255) & (gb =. 255)
let floor = floorNoDoor & (!door)
let wall = !floor

let mrRed = (r =. 255) & (b =. 0) & (g =. 0)
let mrGreen = (r =. 0) & (b =. 0) & (g =. 255)

let mrX = mrRed | mrGreen

let initial1 = exists(mrRed & ((x =. 1) & (y =. 1)))
    .&. exists(mrGreen & ((x =. 1) & (y =. 2)))
let initial2 = exists(mrRed & ((x =. 6) & (y =. 3)))
    .&. exists(mrGreen & ((x =. 1) & (y =. 4)))
let wrong = exists(mrX & wall) .|. (!. (exists(mrRed)
    .&. exists(mrGreen)))
let exit = (x =. 9) & (y >. 2) & (y <. 9)
let pathToExit = (floor ~> exit)
let canExit(mr) = forallin(mr,pathToExit)
let sameRoom = forallin(mrGreen,(mrGreen|floor) ~> mrRed)

let greenFlees = (!.wrong) .&. canExit(mrGreen)
    .&. (!.(canExit(mrRed)))
let nearby = exists(mrRed & (N N mrGreen))

let forbidden1 = greenFlees .|. wrong .|. (!. nearby)
let final1 = exists(mrRed & exit)
    .&. exists(mrGreen & exit)
let final2 = greenFlees .|. wrong
let forbidden2 = final1
```

Fig. 7 `VoxLogicA` specification of the properties of experiments 1 and 2

The most permissive synthesised strategy consists of 684 states and 2635 transitions (recall that in each transition, only one of the agents is moving). The length of a shortest path from the initial state to a final state is composed of 33 transitions to be executed. In the initial state (Figure 6 (left)), the green agent is in front of the red agent on the path to the exit. However, in the strategy the green agent cannot traverse the open door before the red agent. Indeed, in this case, since the door is uncontrollable, it is not possible to prevent the door from closing and separating the red agent (blocked by the door) from the green agent (who can reach the exit). This is indeed a forbidden state that the strategy must avoid. In the strategy, to overcome this problem, the two agents switch position before crossing the door. Figure 6 (right) depicts the moment where the red agent is crossing the door right after exchanging position with the green agent who is still in the corridor. Indeed, in the shortest path they switch position near the door. Note that no forbidden state occurs if the door closes after only the red agent has traversed it. Indeed, in this scenario the green agent is prevented from reaching an exit because it is blocked by the door. Hence, after the red agent has traversed the door, the strategy guides the green agent to safely cross the door such that they can both reach a final state.

To confirm the first experiment, we performed a second experiment by inverting the setup of the first experiment. In this second experiment, the door is controllable, whereas the green and red agents are both uncontrollable. The final states are those in which the door separates the green agent (on the right side of the door) from the red agent (on the left side of the door). These are basically the forbidden states of the first experiment. Similarly, the forbidden states in the second experiment are those states in which both the green and the red agent have reached the exit, i.e., the final states of the first experiment. The initial configuration is the same as in the first experiment. As expected, in this dual case the returned strategy is empty. Indeed, if this were not the case, then we would have a contradiction because the green and red agents have a strategy to reach the exit without being separated by the door with the red agent blocked, for every possible finite behaviour of the door.

There is no strategy for the door to reach a final configuration mainly because the door cannot ensure that the uncontrollable green agent traverses the door first. Moreover, the door cannot prevent the agents from reaching the exit by always remaining closed since (unless only the green agent has traversed the door) a final state would not be reachable.

4.2 Railway Example

The third experiment is based on a real-world case study from the railway domain, which is a well-known application domain for formal methods [51]. The specific example is taken from [23] and it is displayed in Figure 8. Similarly to the previous two experiments, this experiment uses two agents and a gate. It serves as a proof-of-concept of the practical applicability of the toolchain to real-world problems.

In [23], an autonomous tram positioning (ATP) system is analysed. In an ATP, the physical track circuits detecting the occupancy of portions of the railway track are substituted by virtual track circuits (VTCs). The VTCs are virtual positions on a map. The real position of a train is detected using a global positioning system.

In the scenario in Figure 8, one junction area (commanded by one interlocking) is composed of two VTCs, and there is one tram outside the junction area and one tram inside the junction area. Tram 2 is traversing its assigned route while Tram 1 is waiting at a red signal for its route to be assigned. VTC 0 is used to detect the occupation of a route, whereas VTC 1 is used

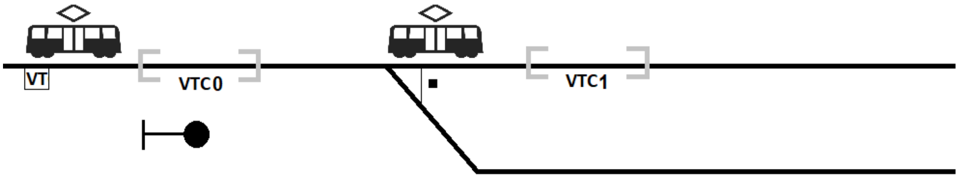

Fig. 8 The scenario taken from [23], where one train is waiting to enter a junction area while another train is traversing it

to detect the release of a route. Initially, both trains are located behind the semaphore. The first train will communicate its route to the interlocking, which will proceed to set the route. This may cause the movement of the junction point. Once the route is set, the interlocking will signal to the train that the route is set by opening the semaphore. The train enters the junction point and the semaphore is closed again. While the first train is traversing its route, the second train will stop at the (closed) semaphore to ask for its route. The route will be assigned, the junction point moved, and the semaphore opened only after the first train has exited the junction area. Otherwise, the movement of the junction point could cause the derailment of the train inside the junction area [23].

We have modelled this scenario using images. Indeed, in general, maps of railway stations are bidimensional planimetries (generally called centralised traffic control boards). In this experiment, the `PNG` map image used as planimetry is a 11×6 pixels image that weighs 157 bytes. This image is derived from the junction scenario in Figure 8. The image representing the initial configuration of this experiment is depicted in Figure 9 (left). Initially, both trains (green and red) are waiting for entering the junction area, whose access is signalled by a semaphore (blue). The semaphore acts like the gate in the maze examples. The open gate models the green signal, whilst the closed gate models the red signal. The VTC 0 detecting the occupancy of the junction area is at the right side of the semaphore, while we assume that the last two pixels on the right borders of the image both contain VTCs for realising the junction. Figure 9 (middle) shows a final configuration where both trains exited the junction area from the same exit, whereas Figure 9 (right) shows in purple the junction area.

We want to synthesise a controller for the semaphore such that both trains can safely traverse the junction area. Therefore, we synthesise a strategy for a game in which the semaphore is the player and the opponents are the trains. The successful final states are those where both trains have exited the junction area, i.e., they both reach the right side of the image. In this case, the trains cannot travel backwards (from right to left). This means that in the composition, the `Driver` automaton (cf. Figure 3) does not have the transition labelled with `!goleft`. The forbidden states are those where both trains are inside the junction area as well as those where one train is inside the junction area, the other train is before the semaphore, and the semaphore is opened (thus creating a hazardous scenario).

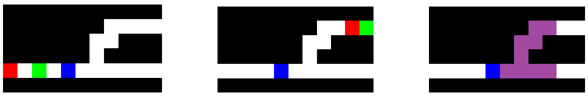

Fig. 9 On the left, the initial configuration of the railway example. In the middle, a final configuration where both trains exit the junction from the same route. On the right, the junction area is emphasised in purple.

Spatial Properties. The setting in this experiment is quite similar to that of the previous ones. The semaphore changes its colour dynamically, and this requires to check which part of the floor is free (`floor` is the white area in the specification) for each image, differently than in the previous examples (cf. Figures 7 and 10). We are also interested in knowing where the semaphore is (`gate` in the specification). We then specify what is the junction area, namely, the area where the railway forks after the semaphore: this is a dangerous area. Then we consider the area before the semaphore (`out` in the specification) and the exits (`outOfJunction`) to be safe. The semaphore is green in those images where there is no blue pixel in $(4, 2)$: we can thus match these coordinated with the floor (note that it is not needed to explicitly check for the `y` coordinate, as it is implicitly assumed by the floor). Trains, which are represented by the green and the red pixels, are defined as `MrGreen` and `MrRed`, as in the previous specification.

At this point, we can start to define our properties of interest. The possible final combinations are represented by those in which both trains reach the exit. So we have two possibilities: `MrRed` reaches the upward

```
let r = red(img)
let g = green(img)
let b = blue(img)

let exists(p) = volume(p) .>. 0
let forall(p) = volume(p) .=. volume(tt)
let forallin(x,p) = forall( (!x) | p)

let gate = (r =. 0) & (b =. 255) & (g =. 0)
let floorNoDoor = (r =. 255) & (b =. 255) & (g =. 255)
let floor = floorNoDoor & (!gate)
let wall = !floor

let mrRed = (r =. 255) & (b =. 0) & (g =. 0)
let mrGreen = (r =. 0) & (b =. 0) & (g =. 255)

let mrX = mrRed | mrGreen

let initial = exists(mrRed & ((x =. 0) & (y =. 4)))
    .&. exists(mrGreen & ((x =. 2) & (y =. 4)))
let junction = (x >. 4) & (x <. 9)
let out = (x >.0) & (x <. 4)
let outOfJunction = (x >. 8) & (x <. 11)
let gateOpen = (floor & x =. 4)
let gOp1 = exists(gateOpen) .&. exists(mrRed & out)
    .&. exists(mrGreen & junction)
let gOp2 = exists(gateOpen) .&. exists(mrGreen & out)
    .&. exists(mrRed & junction)
let wrongSignal = gOp1 .|. gOp2

let combination1 = exists(mrRed & outOfJunction)
    .&. exists(mrGreen & outOfJunction)
let combination2 = exists(mrGreen & outOfJunction)
    .&. exists(mrRed & outOfJunction)

let final = combination1 .|. combination2

let forbidden = (exists(mrRed & junction)
    .&. exists(mrGreen & junction))
    .|. wrongSignal
```

Fig. 10 VoxLogicA specification of the properties of experiment 3

exit, while mrGreen reaches the other exit, or vice versa. Hence, the final state is the disjunction of the two.

We can now focus on the forbidden state. Also in this case, we have two possibilities, but things are a bit more involved. Indeed, we may have a forbidden state in the case where only one of the trains is inside the junction area, but the semaphore is green. As previously said, this could lead to a dangerous situation, and must thus be avoided. This situation is managed by defining the two properties gOp1 and gOp2. The property wrongSignal, which is the disjunction of the two, is the first part of our forbidden property. The other forbidden situation is represented by the simpler case of having both trains inside the junction. This is managed directly in the definition of the forbidden property, by means of the conjunction (exists(mrRed & junction) .&. exists(mrGreen & junction)). In the end, the forbidden property is defined as the disjunction of the latter and wrongSignal.

Synthesis. In the previous experiments, the player transitions were controllable, whereas the opponent transitions were uncontrollable (called *urgent* necessary transitions in contract automata [16]), and the standard most permissive controller synthesis of supervisory control theory (implemented in CATLib) was used to synthesise the strategies. However, the standard notion of uncontrollability from supervisory control theory is too strict for this third experiment. This is because in the composed automaton, the transitions of the uncontrollable opponents (i.e., the trains) would be prioritised over those of the controllable player (i.e., the semaphore). This causes the strategy to be empty. Indeed, the semaphore must be opened to reach a final state (both trains have crossed the junction area). Once the semaphore is opened, both opponent trains can execute a sequence of uncontrollable transitions leading them to a forbidden state (e.g., both trains inside the junction area). These uncontrollable transitions have higher priority than the player's controllable transitions (i.e., closing the semaphore). Indeed, the setting of a player (the semaphore) playing against an opponent (the trains) is not suitable for this particular case study.

For this reason, the *orchestration* synthesis introduced in contract automata theory is used instead of the mpc synthesis (cf. Section 2). Differently from the mpc synthesis, in the orchestration synthesis, the transitions are partitioned into controllable for the player and semi-controllable (a.k.a. *lazy* necessary transitions) for the opponent [16]. The notion of semi-controllability emerges when dealing with concurrent compositions of agents who can internally decide their next transition, but whose scheduling is controllable [8]. In this experiment, the player controls both the semaphore and also decides whose service in the composition will execute the next transition (i.e., the player also acts as the scheduler/orchestrator). All transitions (i.e., the trains' movements) of the the opponent cannot be prevented from eventually being executed, but their execution can be scheduled to happen after the execution of transitions of other services in the composition (i.e., the semaphore transitions). This means that the (controllable) semaphore can thus be closed before the (semi-controllable) second train traverses it. Indeed, the trains are not competing against the semaphore, but are rather cooperating with the semaphore to traverse their (internally chosen) route safely.

The synthesised strategy consists of 180 states and 469 transitions. In this strategy, once the semaphore is opened and the first train enters the junction area, the closing of the semaphore happens prior to the second train entering the junction area. After the first train has exited the junction area, the semaphore is opened again, and the second train can now execute its transitions to reach the exit of the junction area. Hence, a final state is reached without traversing forbidden states.

We conclude with some further remarks on the difference between semi-controllability and controllability. Note that in Figure 9 (right), on both routes the two right-most pixels are outside the junction area. This allows both trains to exit the junction area from the same route, as shown in Figure 9 (middle). Conversely, if only the right-most pixel were outside the junction area, then it would not be possible for both trains to exit the junction from the same route. The trains could only exit the junction area if their route is different. In this case, a non-empty strategy is only possible if also the trains are controllable (i.e., there is no opponent). Instead, if the trains are semi-controllable, then there would be no strategy to guarantee that both trains eventually exit the junction area. This is because the opponent decides internally which route each train will take. Hence, the player cannot prevent the trains from taking the same route.

4.3 Performance of the Experiments

We now report the time needed to compute various phases of the three experiments and measures of the computed automata. We also discuss the performance improvement with respect to [10]. The experiments have been performed on a machine with Intel(R) Core(TM) i9-9900K CPU @ 3.60 GHz equipped with 32 GB of RAM. This is the same machine that was used in [10].

The time performance is reported in Table 3. We note that the synthesis is (computationally) more expensive than the composition. Indeed, as showed in Section 2, each iteration of the synthesis requires to compute the set of dangling states, which requires a forward and backward visit of the automaton. The performances of the synthesis are similar to those of [10]. There is a slight difference due to the updated version of `CATLib` (v.1.0.2) that has been used in this paper with respect to the one used in [10] (v.1.0.1). In `CATLib` v.1.0.2, the forward and backward visit of the synthesis has been updated from a (tail) recursive schema to an iterative schema. This solved the problem of requiring a larger than default size of the stack to avoid stack overflow errors due to the many recursive calls.

We also analysed the improvement in the various activities of `CATLib` with respect to [10]. We only compare the performance of the first two experiments (Table 1), since the third experiment was not present in [10].

In [10], the composition was computed twice, first with all possible behaviour and subsequently with only legal behaviour. This required a total of 29130 ms, which has been reduced to 1739 ms in this paper. Similarly, the generation of images has been improved by passing from 7910 ms in [10] to 1518 ms in this paper. This is because the number of images to generate (i.e., the number of states in the composition) has been reduced. Finally, we consider the activity of marking the composition with the properties computed by `VoxLogicA`. Again, the log to parse in this paper is much smaller than the one that was produced in [10], due to the reduced number of images that are generated. In [10], the marking required 108058 ms and 118291 ms for, respectively, the first and the second experiment. In this paper, the time has been reduced to 29822 ms and 10161 ms, respectively.

Finally, concerning the performance of the third experiment in Table 2, we note that it is better (faster) than either of the first two experiments, mainly due to the smaller size of the image.

Tables 4 and 5 report the number of states, the number of transitions, and the size (in bytes) of the various automata. As expected, the number of states of the agent automata equals the number of pixels of the images. We note that an automaton encoding an image weighs more than the starting `PNG` image. Finally, the marked compositions have two additional states with respect to the composition, which are the added initial and final states. The number of transitions of these two automata differs according to the number of states marked as final, to which a transition to the newly added final state is added, and the number of forbidden states, to which a bad transition is added as a self-loop.

Spatial Properties. The analysis of spatial properties is performed using the image analyser `VoxLogicA`, as in [10]. However, the overall performance has improved significantly: indeed, the execution time in the previous version of the toolchain was dominated by the overhead of setting up the computation, importing the images and parsing the specification several times. This depends on the fact that a single image at a time was processed. In this paper, we overcome this issue by adopting a *batching* technique, similar to the one used in [31]. The logical specification is parsed by a `python` script, which automatically generates a new, much larger specification, replicating the original specification k times, each time replacing the name of the image to be analysed, so that k images are processed in a single run. `VoxLogicA` is then called to process the newly generated specification, and to produce the `JSon` script to be used by `CATLib`. The size k of each such 'batch' is limited to 150 images, as the current version of `VoxLogicA` does not include a garbage collector (which is a planned feature), therefore much larger specifications may result in memory overflows. By using this technique, the overhead for loading and starting the tool's executable, and parsing the specification, only affects the computation time once every k runs of the model checker, instead of

Table 3 Time needed to perform the three experiments

Experiment Phase	First and Second		Third
Computing the composition	1739 ms		633 ms
Generating images	1518 ms		345 ms
	First	**Second**	
Running VoxLogicA	84379 ms	87008 ms	6594 ms
Marking the composition with VoxLogicA properties and controllability	29822 ms	10161 ms	691 ms
Synthesis	31391 ms	11979 ms	1411 ms

Table 4 Number of states, number of transitions, and size of the automata used in the first two experiments

Automaton	#States	#Transitions	Size (bytes)
Agent	100	360	19087
Composition	3200	15176	2019916
Marked composition (first experiment)	3202	17651	2355749
Marked composition (second experiment)	3202	15549	2081535
Strategy (first experiment)	684	2635	350327

Table 5 Number of states, number of transitions, and size of the automata used in the third experiment

Automaton	#States	#Transitions	Size (bytes)
Train	66	230	12269
Composition	363	1084	145485
Marked composition	365	1232	165388
Strategy	180	469	62962

each time. Furthermore, the degree of parallelism can be much higher, as the processing of each image is independent from the others, and all such threads are executed concurrently on multi-core machines.

5 Conclusion

We have discussed a further integration of the tools `CATLib` and `VoxLogicA` to perform strategy synthesis on images processed with spatial model checking.

Our original contribution, in [10], constituted the first application of `CATLib` and `VoxLogicA` to build a framework for modelling and solving mobility problems of collective multi-agent systems, introducing an original approach to combine strategy synthesis with spatial model checking. In this paper, we present a full-fledged toolchain built from `CATLib` and `VoxLogicA`, thus enriching the available tooling for modelling and analysis of CAS. We have improved the efficiency of the encodings, the computations, and the tool integration. The preliminary experiments we performed in [10] have been enriched with a realistic example based on a case study from the railway domain. Several interesting opportunities for future work remain.

Future Work. The toolchain could be improved further. In the current approach, each agent has a state for each pixel of the image. Relaxing the representation of an image to one where each state is a zone of the image (e.g., a corridor) rather than a pixel would reduce the state space. Another improvement could be to decompose a large image into smaller images. For instance, the final states of the first experiment in Section 4 could be entering points to a new portion of the map. Several small maps could be linked together by ports for entering and exiting. Yet another improvement could be to drop the requirement of a strategy to be most permissive in favour of some objective function to optimise. A near-optimal solution could be synthesised as a trace using statistics over runs, in the style of [55]. As for the possibility of allowing more than two agents, this would lead to an exponential blow up of the state space of the composed automaton. However, in future work we plan to exploit state-space reduction techniques, like the ones just discussed, to tackle this kind of problem.

Concerning metrics, from the behavioural point of view, the purpose of our analysis is qualitative, i.e., to synthesise strategies described through finite state automata that enforce qualitative properties described in the spatial logic supported by `VoxLogicA`. Future work is needed for synthesising strategies enforcing quantitative properties. From the spatial point of view, the language of `VoxLogicA` already supports distance and texture similarity measures, which could definitely be employed in further examples similar to the ones included in this paper. Finally, concerning `VoxLogicA`, the input/output overhead could also be eliminated by exploiting the recent GPU implementation [30], which is expected to yield an additional speedup, typically of 1–2 orders of magnitude. A future version of the toolchain is meant to leverage on such currently experimental improvements.

Acknowledgements This study was initiated within the PRIN 2017FTXR7S project IT MaTTerS (Methods and Tools for Trustworthy Smart Systems), which was funded by the Italian Ministry for Universities and Research (MUR).

Part of this study was carried out within the MOST – Sustainable Mobility National Research Center and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1033 17/06/2022, CN00000023) and from the European Union – Next Generation EU – Italian MUR Project PNRR PRI ECS00000017 PRR.AP008.003 "THE – Tuscany Health Ecosystem". This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

CRediT author statement D. Basile: Conceptualization, Software, Formal Analysis, Investigation, Writing - Original Draft, Writing - Review & Editing. M.H. ter Beek: Writing - Original Draft, Writing - Review & Editing, Supervision, Funding Acquisition, Project Administration. L. Bussi: Software, Writing - Review & Editing. V. Ciancia: Conceptualization, Software, Formal Analysis, Investigation, Writing - Original Draft, Writing - Review & Editing.

References

1. Alsalehi, S., Mehdipour, N., Bartocci, E., Belta, C.: Neural Network-based Control for Multi-Agent Systems from Spatio-Temporal Specifications. In: Proceedings of the 60th IEEE Conference on Decision and Control (CDC 2021), pp. 5110–5115. IEEE (2021). doi:10.1109/CDC45484.2021.9682921
2. Alur, R., Moarref, S., Topcu, U.: Compositional Synthesis of Reactive Controllers for Multi-agent Systems. In: S. Chaudhuri, A. Farzan (eds.) CAV 2016, *LNCS*, vol. 9780, pp. 251–269. Springer (2016). doi:10.1007/978-3-319-41540-6_14
3. Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller Synthesis for Timed Automata. IFAC Proc. Vol. **31**(18), 447–452 (1998). doi:10.1016/S1474-6670(17)42032-5
4. Banci Buonamici, F., Belmonte, G., Ciancia, V., Latella, D., Massink, M.: Spatial logics and model checking for medical imaging. Int. J. Softw. Tools Technol. Transf. **22**(2), 195–217 (2020). doi:10.1007/s10009-019-00511-9
5. Basile, D., ter Beek, M.H.: A Clean and Efficient Implementation of Choreography Synthesis for Behavioural Contracts. In: F. Damiani, O. Dardha (eds.) COORDINATION 2021, *LNCS*, vol. 12717, pp. 225–238. Springer (2021). doi:10.1007/978-3-030-78142-2_14
6. Basile, D., ter Beek, M.H.: Contract Automata Library. Sci. Comput. Program. **221** (2022). doi:10.1016/j.scico.2022.102841. URL https://github.com/contractautomataproject/ContractAutomataLib
7. Basile, D., ter Beek, M.H.: A Runtime Environment for Contract Automata. In: M. Chechik, J.P. Katoen, M. Leucker (eds.) FM 2023, *LNCS*, vol. 14000, pp. 550–567. Springer (2023). doi:10.1007/978-3-031-27481-7_31
8. Basile, D., ter Beek, M.H.: Research Challenges in Orchestration Synthesis. In: C. Aubert, C. Di Giusto, S. Fowler, L. Safina (eds.) Proceedings of the 16th Interaction and Concurrency Experience (ICE 2023), *EPTCS*, vol. 383, pp. 73–90 (2023). doi:10.4204/EPTCS.383.5
9. Basile, D., ter Beek, M.H., Bussi, L., Ciancia, V.: A Toolchain for Strategy Synthesis with Spatial Properties - Complementary Material (2023). doi:10.5281/zenodo.8220528
10. Basile, D., ter Beek, M.H., Ciancia, V.: An Experimental Toolchain for Strategy Synthesis with Spatial Properties. In: T. Margaria, B. Steffen (eds.) ISoLA 2022, *LNCS*, vol. 13703, pp. 142–164. Springer (2022). doi:10.1007/978-3-031-19759-8_10
11. Basile, D., ter Beek, M.H., Degano, P., Legay, A., Ferrari, G.L., Gnesi, S., Di Giandomenico, F.: Controller synthesis of service contracts with variability. Sci. Comput. Program. **187** (2020). doi:10.1016/j.scico.2019.102344
12. Basile, D., ter Beek, M.H., Di Giandomenico, F., Gnesi, S.: Orchestration of Dynamic Service Product Lines with Featured Modal Contract Automata. In: Proceedings of the 21st International Systems and Software Product Line Conference (SPLC 2017), vol. 2, pp. 117–122. ACM (2017). doi:10.1145/3109729.3109741
13. Basile, D., ter Beek, M.H., Gnesi, S.: Modelling and Analysis with Featured Modal Contract Automata. In: Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC 2018), vol. 2, pp. 11–16. ACM (2018). doi:10.1145/3236405.3236408
14. Basile, D., ter Beek, M.H., Legay, A.: Strategy Synthesis for Autonomous Driving in a Moving Block Railway System with UPPAAL STRATEGO. In: A. Gotsman, A. Sokolova (eds.) FORTE 2020, *LNCS*, vol. 12136, pp. 3–21. Springer (2020). doi:10.1007/978-3-030-50086-3_1
15. Basile, D., ter Beek, M.H., Legay, A.: Timed service contract automata. Innovations Syst. Softw. Eng. **16**(2), 199–214 (2020). doi:10.1007/s11334-019-00353-3
16. Basile, D., ter Beek, M.H., Pugliese, R.: Synthesis of Orchestrations and Choreographies: Bridging the Gap between Supervisory Control and Coordination of Services. Log. Methods Comput. Sci. **16**(2) (2020). doi:10.23638/LMCS-16(2:9)2020
17. Basile, D., Degano, P., Ferrari, G.L.: Automata for Specifying and Orchestrating Service Contracts. Log. Methods Comput. Sci. **12**(4) (2016). doi:10.2168/LMCS-12(4:6)2016
18. Basile, D., Degano, P., Ferrari, G.L., Tuosto, E.: Playing with Our CAT and Communication-Centric Applications. In: E. Albert, I. Lanese (eds.) FORTE 2016, *LNCS*, vol. 9688, pp. 62–73. Springer (2016). doi:10.1007/978-3-319-39570-8_5
19. Basile, D., Degano, P., Ferrari, G.L., Tuosto, E.: Relating two automata-based models of orchestration and choreography. J. Log. Algebr. Methods Program. **85**(3), 425–446 (2016). doi:10.1016/j.jlamp.2015.09.011
20. Basile, D., Di Giandomenico, F., Gnesi, S.: Enhancing Models Correctness through Formal Verification: A Case Study from the Railway Domain. In: Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), pp. 679–686. SciTePress (2017). doi:10.5220/0006291106790686

21. Basile, D., Di Giandomenico, F., Gnesi, S.: FMCAT: Supporting Dynamic Service-based Product Lines. In: Proceedings of the 21st International Systems and Software Product Line Conference (SPLC 2017), vol. 2, pp. 3–8. ACM (2017). doi:10.1145/3109729.3109760
22. Basile, D., Di Giandomenico, F., Gnesi, S., Degano, P., Ferrari, G.L.: Specifying Variability in Service Contracts. In: Proceedings of the 11th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2017), pp. 20–27. ACM (2017). doi:10.1145/3023956.3023965
23. Basile, D., Fantechi, A., Rucher, L., Mandò, G.: Analysing an autonomous tramway positioning system with the Uppaal Statistical Model Checker. Form. Asp. Comput. **33**(6), 957–987 (2021). doi:10.1007/s00165-021-00556-1
24. ter Beek, M.H., Reniers, M.A., de Vink, E.P.: Supervisory Controller Synthesis for Product Lines Using CIF 3. In: T. Margaria, B. Steffen (eds.) ISoLA 2016, *LNCS*, vol. 9952, pp. 856–873. Springer (2016). doi:10.1007/978-3-319-47166-2_59
25. Belmonte, G., Broccia, G., Vincenzo, C., Latella, D., Massink, M.: Feasibility of Spatial Model Checking for Nevus Segmentation. In: Proceedings of the 9th IEEE/ACM International Conference on Formal Methods in Software Engineering (FormaliSE 2021), pp. 1–12. IEEE (2021). doi:10.1109/FormaliSE52586.2021.00007
26. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: Innovating Medical Image Analysis via Spatial Logics. In: M.H. ter Beek, A. Fantechi, L. Semini (eds.) From Software Engineering to Formal Methods and Tools, and Back, *LNCS*, vol. 11865, pp. 85–109. Springer (2019). doi:10.1007/978-3-030-30985-5_7
27. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: VoxLogicA: A Spatial Model Checker for Declarative Image Analysis. In: T. Vojnar, L. Zhang (eds.) TACAS 2019, *LNCS*, vol. 11427, pp. 281–298. Springer (2019). doi:10.1007/978-3-030-17462-0_16
28. Bernardo, M., De Nicola, R., Hillston, J. (eds.): Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems, *LNCS*, vol. 9700. Springer (2016). doi:10.1007/978-3-319-34096-8
29. Bezhanishvili, N., Ciancia, V., Gabelaia, D., Grilletti, G., Latella, D., Massink, M.: Geometric Model Checking of Continuous Space. Log. Methods Comput. Sci. **18**(4), 7:1–7:38 (2022). doi:10.46298/lmcs-18(4:7)2022
30. Bussi, L., Ciancia, V., Gadducci, F.: Towards a Spatial Model Checker on GPU. In: K. Peters, T.A.C. Willemse (eds.) FORTE 2021, *LNCS*, vol. 12719, pp. 188–196. Springer (2021). doi:10.1007/978-3-030-78089-0_12
31. Bussi, L., Ciancia, V., Gadducci, F., Latella, D., Massink, M.: Towards Model Checking Video Streams Using VoxLogicA on GPUs. In: J. Bowles, G. Broccia, R. Pellungrini (eds.) DataMod 2021, *LNCS*, vol. 13268, pp. 78–90. Springer (2021). doi:10.1007/978-3-031-16011-0_6
32. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding Parity Games in Quasipolynomial Time. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2017), pp. 252–263. ACM (2017). doi:10.1145/3055399.3055409
33. Camacho, A., Bienvenu, M., McIlraith, S.A.: Towards a Unified View of AI Planning and Reactive Synthesis. In: Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2018), pp. 58–67. AAAI (2019). URL https://ojs.aaai.org/index.php/ICAPS/article/view/3460
34. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer (2006). doi:10.1007/978-0-387-68612-7
35. Castelnovo, D., Miculan, M.: Closure Hyperdoctrines. In: F. Gadducci, A. Silva (eds.) Proceedings of the 9th Conference on Algebra and Coalgebra in Computer Science (CALCO 2021), *LIPIcs*, vol. 211, pp. 12:1–12:21 (2021). doi:10.4230/LIPIcs.CALCO.2021.12
36. Cauchi, N., Abate, A.: StocHy: Automated Verification and Synthesis of Stochastic Processes. In: T. Vojnar, L. Zhang (eds.) TACAS 2019, *LNCS*, vol. 11428, pp. 247–264. Springer (2019). doi:10.1007/978-3-030-17465-1_14
37. Ceska, M., Pilar, P., Paoletti, N., Brim, L., Kwiatkowska, M.Z.: PRISM-PSY: Precise GPU-Accelerated Parameter Synthesis for Stochastic Systems. In: M. Chechik, J. Raskin (eds.) TACAS 2016, *LNCS*, vol. 9636, pp. 367–384. Springer (2016). doi:10.1007/978-3-662-49674-9_21
38. Cheng, C., Lee, E.A., Ruess, H.: autoCode4: Structural Controller Synthesis. In: A. Legay, T. Margaria (eds.) TACAS 2017, *LNCS*, vol. 10205, pp. 398–404. Springer (2017). doi:10.1007/978-3-662-54577-5_23
39. Ciancia, V., Belmonte, G., Latella, D., Massink, M.: A Hands-On Introduction to Spatial Model Checking Using VoxLogicA – Invited Contribution. In: A. Laarman, A. Sokolova (eds.) SPIN 2021, *LNCS*, vol. 12864, pp. 22–41. Springer (2021). doi:10.1007/978-3-030-84629-9_2
40. Ciancia, V., Gilmore, S., Grilletti, G., Latella, D., Loreti, M., Massink, M.: Spatio-temporal model checking of vehicular movement in public transport systems. Int. J. Softw. Tools Technol. Transf. **20**(3), 289–311 (2018). doi:10.1007/s10009-018-0483-8
41. Ciancia, V., Gilmore, S., Latella, D., Loreti, M., Massink, M.: Data Verification for Collective Adaptive Systems: Spatial Model-Checking of Vehicle Location Data. In: Proceedings of the 8th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2014), pp. 32–37. IEEE (2014). doi:10.1109/SASOW.2014.16
42. Ciancia, V., Latella, D., Loreti, M., Massink, M.: Model Checking Spatial Logics for Closure Spaces. Log. Methods Comput. Sci. **12**(4) (2016). doi:10.2168/LMCS-12(4:2)2016
43. Ciancia, V., Latella, D., Massink, M., Paškauskas, R.: Exploring Spatio-temporal Properties of Bike-Sharing Systems. In: Proceedings of the Workshops at the 9th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASOW 2015), pp. 74–79. IEEE (2015). doi:10.1109/SASOW.2015.17
44. Ciancia, V., Latella, D., Massink, M., Paškauskas, R., Vandin, A.: A Tool-Chain for Statistical Spatio-Temporal Model Checking of Bike Sharing Systems. In: T. Margaria, B. Steffen (eds.) ISoLA 2016, *LNCS*, vol. 9952, pp. 657–673. Springer (2016). doi:10.1007/978-3-319-47166-2_46
45. David, A., Jensen, P.G., Larsen, K.G., Mikucionis, M., Taankvist, J.H.: Uppaal Stratego. In: C. Baier, C. Tinelli (eds.) TACAS 2015, *LNCS*, vol. 9035, pp. 206–211. Springer (2015). doi:10.1007/978-3-662-46681-0_16
46. Deniélou, P., Yoshida, N.: Multiparty Session Types Meet Communicating Automata. In: H. Seidl (ed.) ESOP 2012, *LNCS*, vol. 7211, pp. 194–213. Springer (2012). doi:10.1007/978-3-642-28869-2_10
47. Ehlers, R., Lafortune, S., Tripakis, S., Vardi, M.Y.: Supervisory control and reactive synthesis: a comparative introduction. Discret. Event Dyn. Syst. **27**(2), 209–260 (2017). doi:10.1007/s10626-015-0223-0

48. Fan, C., Miller, K., Mitra, S.: Fast and Guaranteed Safe Controller Synthesis for Nonlinear Vehicle Models. In: S.K. Lahiri, C. Wang (eds.) CAV 2020, *LNCS*, vol. 12224, pp. 629–652. Springer (2020). doi:10.1007/978-3-030-53288-8_31
49. Farhat, H.: Web Service Composition via Supervisory Control Theory. IEEE Access **6**, 59779–59789 (2018). doi:10.1109/ACCESS.2018.2874564
50. Felli, P., Yadav, N., Sardina, S.: Supervisory Control for Behavior Composition. IEEE Trans. Autom. Control **62**(2), 986–991 (2017). doi:10.1109/TAC.2016.2570748
51. Ferrari, A., ter Beek, M.H.: Formal Methods in Railways: a Systematic Mapping Study. ACM Comput. Surv. **55**(4), 69:1–69:37 (2023). doi:10.1145/3520480
52. Ferscha, A.: Collective Adaptive Systems. In: Adjunct Proceedings of the 19th ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 19th ACM International Symposium on Wearable Computers (UbiComp/ISWC 2015 Adjunct), pp. 893–895. ACM (2015). doi:10.1145/2800835.2809508
53. Forschelen, S.T.J., van de Mortel-Fronczak, J.M., Su, R., Rooda, J.E.: Application of supervisory control theory to theme park vehicles. Discrete Event Dyn. Syst. **22**(4), 511–540 (2012). doi:10.1007/s10626-012-0130-6
54. Glazier, T.J., Cámara, J., Schmerl, B.R., Garlan, D.: Analyzing Resilience Properties of Different Topologies of Collective Adaptive Systems. In: Proceedings of the 9th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2015), pp. 55–60. IEEE (2015). doi:10.1109/SASOW.2015.14
55. Gu, R., Jensen, P.G., Poulsen, D.B., Seceleanu, C., Enoiu, E., Lundqvist, K.: Verifiable strategy synthesis for multiple autonomous agents: a scalable approach. Int. J. Softw. Tools Technol. Transf. **24**(3), 395–414 (2022). doi:10.1007/s10009-022-00657-z
56. Guo, M., Dimarogonas, D.V.: Multi-agent plan reconfiguration under local LTL specifications. Int. J. Robotics Res. **34**(2), 218–235 (2015). doi:10.1177/0278364914546174
57. 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), pp. 189–198. ACM (2015). doi:10.1145/2728606.2728633
58. Haghighi, I., Sadraddini, S., Belta, C.: Robotic Swarm Control from Spatio-Temporal Specifications. In: Proceedings of the 55th IEEE Conference on Decision and Control (CDC 2016), pp. 5708–5713. IEEE (2016). doi:10.1109/CDC.2016.7799146
59. Hillston, J., Pitt, J., Wirsing, M., Zambonelli, F.: Collective Adaptive Systems: Qualitative and Quantitative Modelling and Analysis. Dagstuhl Reports **4**(12), 68–113 (2014). doi:10.4230/DagRep.4.12.68
60. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: PRISM-games 3.0: Stochastic Game Verification with Concurrency, Equilibria and Time. In: S.K. Lahiri, C. Wang (eds.) CAV 2020, *LNCS*, vol. 12225, pp. 475–487. Springer (2020). doi:10.1007/978-3-030-53291-8_25
61. Lange, J., Tuosto, E., Yoshida, N.: From Communicating Machines to Graphical Choreographies. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015), pp. 221–232. ACM (2015). doi:10.1145/2676726.2676964
62. Lavaei, A., Khaled, M., Soudjani, S., Zamani, M.: AMYTISS: Parallelized Automated Controller Synthesis for Large-Scale Stochastic Systems. In: S.K. Lahiri, C. Wang (eds.) CAV 2020, *LNCS*, vol. 12225, pp. 461–474. Springer (2020). doi:10.1007/978-3-030-53291-8_24
63. Lehmann, S., Rogalla, A., Neidhardt, M., Reinecke, A., Schlaefer, A., Schupp, S.: Modeling $\mathbb{R}^3$ Needle Steering in Uppaal. In: C. Dubslaff, B. Luttik (eds.) Proceedings of the 5th Workshop on Models for Formal Analysis of Real Systems (MARS 2022), *EPTCS*, vol. 355, pp. 40–59 (2022). doi:10.4204/EPTCS.355.4
64. Liu, Z., Wu, B., Dai, J., Lin, H.: Distributed Communication-Aware Motion Planning for Networked Mobile Robots Under Formal Specifications. IEEE Trans. Control. Netw. Syst. **7**(4), 1801–1811 (2020). doi:10.1109/TCNS.2020.3000742
65. Loizou, S.G., Kyriakopoulos, K.J.: Automatic Synthesis of Multi-Agent Motion Tasks Based on LTL Specifications. In: Proceedings of the 43rd IEEE Conference on Decision and Control (CDC 2004), pp. 153–158. IEEE (2004). doi:10.1109/CDC.2004.1428622
66. Loreti, M., Quadrini, M.: A Spatial Logic for a Simplicial Complex Model. arXiv:2105.08708 (2021). URL https://arxiv.org/abs/2105.08708
67. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Inform. **57**(1-2), 3–36 (2020). doi:10.1007/s00236-019-00349-3
68. Ma, M., Bartocci, E., Lifland, E., Stankovic, J.A., Feng, L.: A Novel Spatial–Temporal Specification-Based Monitoring System for Smart Cities. IEEE Internet Things J. **8**(15), 11793–11806 (2021). doi:10.1109/JIOT.2021.3069943
69. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit Reactive Synthesis Strikes Back! In: H. Chockler, G. Weissenbacher (eds.) CAV 2018, *LNCS*, vol. 10981, pp. 578–586. Springer (2018). doi:10.1007/978-3-319-96145-3_31
70. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987). doi:10.1137/0325013
71. van der Sanden, B., Reniers, M.A., Geilen, M., Basten, T., Jacobs, J., Voeten, J., Schiffelers, R.R.H.: Modular Model-Based Supervisory Controller Design for Wafer Logistics in Lithography Machines. In: Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), pp. 416–425. IEEE (2015). doi:10.1109/MODELS.2015.7338273
72. Shokri-Manninen, F., Vain, J., Waldén, M.: Formal Verification of COLREG-Based Navigation of Maritime Autonomous Systems. In: F.S. de Boer, A. Cerone (eds.) SEFM 2020, *LNCS*, vol. 12310, pp. 41–59. Springer (2020). doi:10.1007/978-3-030-58768-0_3
73. Sun, D., Chen, J., Mitra, S., Fan, C.: Multi-Agent Motion Planning From Signal Temporal Logic Specifications. IEEE Robotics Autom. Lett. **7**(2), 3451–3458 (2022). doi:10.1109/LRA.2022.3146951
74. Theunissen, R.J.M., van Beek, D.A., Rooda, J.E.: Improving evolvability of a patient communication control system using state-based supervisory control synthesis. Adv. Eng. Inform. **26**(3), 502–515 (2012). doi:10.1016/j.aei.2012.02.009
75. Thuijsman, S., Reniers, M.: Supervisory Control for Dynamic Feature Configuration in Product Lines. ACM Trans. Embed. Comput. Syst. (2023). doi:10.1145/3579644
76. Tsigkanos, C., Kehrer, T., Ghezzi, C.: Modeling and Verification of Evolving Cyber-Physical Spaces. In: Proceed-

ings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017), pp. 38–48. ACM (2017). doi:10.1145/3106237.3106299