





# A Hands-on Introduction to Spatial Model Checking using VoxLogicA <sup>\*</sup>

Invited contribution

Vincenzo Ciancia<sup>1</sup> , Gina Belmonte<sup>2</sup> , Diego Latella<sup>1</sup> , and Mieke Massink<sup>1</sup> 

<sup>1</sup> Istituto di Scienza e Tecnologie dell’Informazione “A. Faedo”,  
Consiglio Nazionale delle Ricerche  
{name.surname}@isti.cnr.it

<sup>2</sup> Azienda Toscana Nord Ovest, S. C. Fisica Sanitaria Nord – Lucca, Italy  
gina.belmonte@uslnordovest.toscana.it

**Abstract.** This paper provides a tutorial-style introduction, and a guide, to the recent advancements in spatial model checking that have made some relevant results possible. Among these, we mention fully automated segmentation of regions of interest in medical images by short, unambiguous spatial-logical specifications. This tutorial is aimed both at domain experts in medical imaging who would like to learn simple (scripting-alike) techniques for image analysis, making use of a modern, declarative language, and at experts in Formal Methods in Computer Science and Model Checking who would like to grasp how the theory of Spatial Logic and Model Checking has been turned into logic-based, dataset-oriented imaging techniques.

**Keywords:** Spatial Logic · Model Checking · Tutorial.

## 1 Introduction

The topological approach to spatial model checking was introduced in [16,17], as a fully automated method to verify properties of points in a spatial structure, such as a graph, or a digital image. The theory of spatial model checking has its roots in the spatial interpretation of modal logics dating back to Tarski (see [9] for a thorough introduction to the subject). Spatial properties of points are related to topological aspects such as being *near* to points satisfying a given property, or being able to *reach* a point satisfying a certain property, passing

---

<sup>\*</sup> Research partially supported by the MIUR PRIN 2017FTXR7S “IT-MaTTerS”.

This tutorial is meant to complement the invited talk in the *27th International SPIN Symposium on Model Checking of Software* by Vincenzo Ciancia, therefore listed as the first author. All the authors equally contributed to developments of the presented research line and are primary authors of this paper.

This article is a preprint. The final authenticated version is available online at [https://doi.org/10.1007/978-3-030-84629-9\\_2](https://doi.org/10.1007/978-3-030-84629-9_2).

only through points obeying to specific constraints. The *Spatial Logic of Closure Spaces* defined in [16] is quite expressive, which has been demonstrated in case studies ranging from smart transportation [19] to bike sharing [23,18], to medical image analysis [3,7,8,6]. The arbitrary nesting of spatial properties is the key to obtain such strong capabilities.

In the tool VoxLogicA, presented in [8], and designed from scratch for image analysis, logical operators can be freely mixed with a few imaging operators, related to colour thresholding, texture analysis, or normalization. The tool is quite fast, due to various factors: most primitives are implemented using the state-of-the-art imaging library SimpleITK<sup>3</sup>; expressions are never recomputed (reduction of the syntax tree to a directed acyclic graph is used as a form of *memoization*); operations are implicitly parallelised on multi-core CPUs. Case studies such as brain tumour segmentation [3,8], labelling of white and grey matter [7], and contouring of nevi [6] have shown that simple, unambiguous and explainable logical specifications can compete in accuracy with state-of-the-art machine-learning based methods<sup>4</sup>.

So far, however, even if the VoxLogicA approach is meant to be domain-expert-friendly and the tool itself is quite straightforward to use, applications of spatial model checking have been confined to a limited group of “initiated” collaborators. One reason for this is a conceptual gap between the theory of topological spatial logics for discrete structures, presented in [16,17], and the technicalities of a full case study such as that of [6], where the most relevant keywords are *dataset*, *overlay*, *ground truth*, *region of interest*, etc.

In this paper, we attempt to fill this gap by providing a gentle, hands-on introduction to the subject of spatial model checking for image analysis. The intended audience of this paper is two-fold: we aim at reaching both domain experts in image analysis (who could even be non-programmers, but are willing to get acquainted with the benefits of declarative analysis, and learn simple scripting-alike techniques to automatise imaging tasks, based on spatial features of points or regions) and experts in Formal Methods in Computer Science, and in particular in model checking, who are able to understand the technical aspects of VoxLogicA, but need some guidance to gather insights from the ideas behind its image analysis capabilities.

In Section 2, we introduce the spatial model checker VoxLogicA starting from the practicalities: how to invoke the tool, the format of input files, visualization of results, log files, and how to run the tool against a dataset. In Section 3, we illustrate by examples the core capabilities of spatial-logical reasoning, that is, the concepts of *nearness* and *reachability*. In Section 4, we introduce the use of VoxLogicA as a method to obtain numbers or Boolean values from whole images,

<sup>3</sup> See <https://simpleitk.org/>.

<sup>4</sup> Indeed, it is not the intention of the research line around VoxLogicA to compete against machine-learning based approaches. Rather, we expect that the two can be complementary: VoxLogicA specifications can certainly be used to coordinate various machine-learning based steps in order to obtain procedures that have a degree of machine-learning based operation, but are still modifiable and explainable.

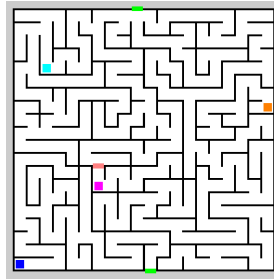
for instance in order to query datasets to find images satisfying given properties. In Section 5, we illustrate some slightly more advanced examples of spatial operators, including *distance transform* and *filtering*, via a background segmentation example. Therein, we also briefly discuss a *statistical texture similarity* operator. In Section 6, we provide a brief guide to the related literature, focusing on recent applications of Spatial Logics in Computer Science, both based on the Spatial Logics of Closure Spaces proposed in [16,17] and not depending upon it. In Section 7, we illustrate the more recent research lines that are being pursued by the VoxLogicA group, including the use of *GPU computing* to speed up spatial model checking, the study of a dataset oriented graphical user interface for the design of logical specifications, and a freshly designed spatial model checker for analysing *3D meshes* using the Spatial Logic of Closure Spaces.

## 2 Using VoxLogicA: Practicalities

In this tutorial, we will use the command line interface of VoxLogicA<sup>5</sup>. After unpacking / installing the tool, running the executable from its full path, with no arguments, will produce a help message listing the options. VoxLogicA specifications contain a description of the *model* to be analysed (essentially, a set of images of the same size), and a list of properties to be verified. For executing a specification, only one argument is needed, that is, the specification name, so the model checker can be run as follows:

```
/path/to/VoxLogicA input.imgql
```

In our first examples, we will use the following image of a maze, with green exit, white corridors, black walls, four coloured square placeholders (cyan, orange, magenta, blue), and a pink trapdoor.



It is very important to remark that although for simplicity, in this tutorial we encode all properties of interest as colours (and doing so can be useful in several situations), real-world examples may require more precise annotation of

<sup>5</sup> We use VoxLogicA version 1.0. It can be downloaded from <https://github.com/vincenzoml/VoxLogicA>. The example images and files of this paper are available at the same web site.

properties of pixels. In medical images, for instance, it is quite common that a dataset contains, for each case, several separated, possibly overlapping images of the same size, either Boolean valued, or divided into regions by the use of integer labels. Such annotations are called *regions of interest* (ROIs). Indeed, VoxLogicA can load more than one image by using several **load** instructions, and thus easily work with multiple ROIs. Similarly, a dataset may contain more than one “base” image. For example, a dataset related to brain tumours could contain acquisitions made with different MRI modalities, which emphasize different aspects of a patient situation. Analysis methods that can make use of multiple modalities are called *multi-modal*. Again, since VoxLogicA can load multiple images in the same specification, it can be readily used for multi-modal analysis.

## 2.1 The declarative language ImgQL

The input language of VoxLogicA, namely the *Image Query Language* ImgQL, has only five commands:

- **load** `x = "filename.{png,nii,nii.gz,jpg,bmp,...}"`  
loads an image in one of the supported file formats, and binds its to the (constant) name `x`. Note that ImgQL is a “pure” language, with no side effects. Therefore all names are constant, not variables (there is no assignment).
- **save** `"filename.{png,nii,nii.gz,jpg,bmp,...}" expression`  
saves the result of an expression returning an image<sup>6</sup> to a file;
- **print** `"label" expression`  
prints the result of an expression returning a number, or a boolean value, to the log file, accompanied by a given label to be easily recognisable;
- **let** `name = expression`  
where `name` starts with a lowercase letter, declares a constant; the **let** construct has two more variants, described below;
- **let** `fname(x1,...,xn) = expression`  
where `fname` starts with a lowercase letter, declares a function;
- **let** `OP(x1,...,xn) = expression`  
where `OP` consists of uppercase letters and symbols, declares an *operator*, which is different from a function only by the syntax used to invoke it. Unary operators are prefix (e.g. the *not/complement* operator `!x`); binary operators are infix (e.g. the *or/union* operator `x & y`); if more than two arguments are supplied, these are added in square brackets; for instance, **let** `OP(x,y,z,t) = ...` defines the operator `OP` invoked as `x OP[z,t] y`;
- **import** `"filename.imgql"`  
imports a library of **let** declarations; no other command than **let** can appear in an imported file. The file `stdlib.imgql` located in the same directory as the VoxLogicA executable is automatically imported. Files are first

<sup>6</sup> A VoxLogicA expression may return either an image – which can be Boolean-valued, number-valued, or have multiple number-valued channels – or a single value, which can be either a number or a Boolean value. No distinction is made between integer and floating point numbers; all numbers are treated as floating point internally.

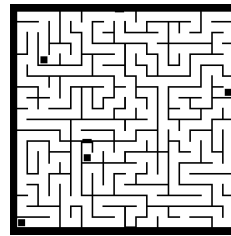
searched in the same directory as the executable. If the file name contains no extension and the file is not found, VoxLogicA also attempts to load a file with the same name and `.imgql` extension.

## 2.2 Loading and saving models

We shall first demonstrate **load** and **save** constructs. We will load our maze image, and save an output image only containing its routes, that is, the white area. Below, the specification is shown on the left, and the result on the right. In white, the pixels satisfying the whole specification, which as we shall see (and purely coincidentally, in this example) are the white pixels of the input image.

```
load img = "maze01.png"
let corridor = (red(img) =. 255) &
               (green(img) =. 255) & (blue(img) =. 255)

save "output/example01.png" corridor
```



The **load** instruction binds to name `img` the image contained in the input file. The **save** instruction saves to the output file the result of an expression (we have bound the expression to the name `corridor`, but that would not be necessary in principle; the expression could have directly appeared as an argument of the **save** instruction). Throughout this tutorial, we save all our output files in a directory named “output”. This is convenient, but indeed not mandatory; the file would be saved in the *current working directory* if no path was supplied. If output directories are specified, these are automatically created if not existing.

Let us analyse the definition of `corridor`, aimed at selecting only the white pixels in the image. First let us look at the sub-expression `red(img) =. 255`. The expression `red(img)` takes as parameter the image `img`, and returns an image consisting in only the red component of the image (similarly, there are functions **green** and **blue** for the other two components of the RGB colour space). Since the input file is a 8-bits-per-channel image, the result of `red(img)` is a number-valued image, having the same width and height of the original one, containing a numeric value between 0 and 255 in each pixel. Note that the VoxLogicA type system does not distinguish between integers and floating points. Internally, all numbers are 32-bits floating point in order to guarantee precision of the analysis. The infix operator `=.` takes on its left a number-valued image  $i$ , and on its right a number  $n$ . As in some scientific computation languages, the dot in an operator is on the side of numbers, whereas “matrices” (in our case, images) do not have a dot. The result is a Boolean-valued image, containing in each pixel at coordinates  $(x, y)$  the value `true` if the value of  $i$  at the pixel  $(x, y)$  is equal to  $n$ . Similarly, there are operators `>.` (greater than a value), `>=.` (greater or equal than a value) and so on. The infix operator `&` is logical conjunction (“and”) (similarly, there is a disjunction operator `|` (“or”), and

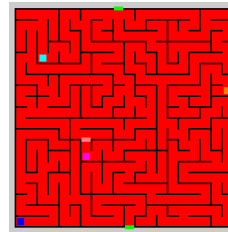
negation ! (“not”). The operator `&` takes as input two Boolean-valued images, and returns a Boolean-valued images which, pixel by pixel, contains the conjunction of the corresponding values. Therefore the meaning of the whole expression `(red(img) =. 255) & (green(img) =. 255) & (blue(img) =. 255)` is to return a Boolean image that has value `true` only on the *white* pixels (all the three components have maximum value).

Since the `save` instruction requires a Boolean valued image to be saved in the `png` format, which only allows integer values, the truth values are encoded as 0 for the value `false` and 255 for the value `true`. This is to ease visualisation of the verification results, as it corresponds to using the colour black for `false` and white for `true`, as it is clearly visible in the resulting image.

Finally, sometimes it is easier to visualise results by super-imposing them on the original image. For this, the `overlayColor` function of the standard library can be of help. It takes as arguments an “overlay” Boolean-valued image (to be super-imposed in a colour), a “background” image (which will be rendered “below” the super-imposed layer), and three colour components, red, green and blue. It returns an image having the pixels coloured in the same way as “background” on the pixels where “overlay” is `false`, and in the colour specified by the three colour components on the pixels where “overlay” is `true`. In the example below, we super-impose in red the expression denoting corridors, on top of our base maze image. In order to do so, we define a single-argument function named `view` that shows its only argument in red, and will be reused later.

```
let view(x) = overlayColor(x, img, 255, 0, 0)

save "output/example02.png" view(corridor)
```



### 2.3 Anatomy of VoxLogicA logs

The log of our first analysis is reported below (path names have been edited).

```
[ 84ms] [info] Parsing input...
[127ms] [info] Preparing computation...
[157ms] [info] Importing file "/path/to/stdlib.imgql"
[167ms] [info] Loading file "/path/to/maze01.png"
[205ms] [warn] image maze01.png has 4 color components per voxel.
           Assuming RGBA color space (CMYK is not supported).
[241ms] [info] Starting computation...
[242ms] [info] Running 10 tasks
[292ms] [warn] saving boolean image to example01.png;
           value 'true' is 255, not 1
[296ms] [info] Saving file "/path/to/output/example01.png"
[334ms] [info] ... done.
```

The log has three columns. The first one contains the time at which each message has been printed (relative to the start of the program). The second column contains the severity level of the message, which can be “**informative**”, “**warning**”, “**failure**”, or “**user**” for messages issued using the **print** instruction. The third column contains the message itself. Among the many possible messages, the number of tasks (in this case, 10) can be useful to estimate the complexity of a formula. It is however worth emphasizing that the number of tasks that will be executed is not directly proportional to the number of subformulas, but rather to the number of *unique* subformulas. The model checking engine of VoxLogicA never computes expressions twice, so the same number of tasks are obtained e.g., by writing `f(expression1,expression1)` or `let x = expression1 and then f(x,x)`. The number of unique expressions depends on reduction of the given formulas to the core primitives of VoxLogicA (which can be listed using the `--ops` command line option).

## 2.4 Working with datasets

One of the major issues related to image analysis using declarative languages is how much *ad-hoc* is resulting specification is. On the other hand, the work in [3,7,8,6] has been successful in carrying out complex imaging tasks, such as brain tumour or nevus segmentation, because “success” is measured against a reasonably large dataset, proving generality of the proposed specification. To the best of our knowledge so far, declarative specifications work best against *homogeneous* datasets<sup>7</sup>. So are, for instance, 3D Magnetic Resonance Images (MRI) obtained using specific MRI modalities, such as the MRI-FLAIR<sup>8</sup> used in [8]. Among many common features, all MRI-FLAIR brain images have a dark background, the cerebrospinal fluid is dark, and the tumour area is always hyper-intense. Indeed such constraint may be relaxed, and effective analysis methodologies can be developed on datasets that are *partitioned* into a number of different, but homogeneous classes, as it happens in [6].

Currently, there is no built-in facility in VoxLogicA to work with datasets, and extract useful information (such as, for instance, performance scores that ought to be optimized). Specifications are meant to be run against single cases. This could be the subject of future improvements to the tool, but currently, it is just easier to resort to an external tool orchestrating several runs of the model checker by replacing file names according to patterns that are defined based on the dataset. For instance, a script for datasets of the Brain Tumour Segmentation challenge [30] is provided in the VoxLogicA repository, and can be readily adapted to datasets with different naming conventions.

<sup>7</sup> In this respect, we consider our work still as the beginning of a research line, and we cannot predict if, for instance, novel logical operators will enable the development of very general specifications that operate on inhomogeneous domains.

<sup>8</sup> Fluid-attenuated Inversion Recovery. See e.g. Wikipedia contributors, “Fluid-attenuated inversion recovery,” Wikipedia, The Free Encyclopedia, [https://en.wikipedia.org/w/index.php?title=Fluid-attenuated\\_inversion\\_recovery](https://en.wikipedia.org/w/index.php?title=Fluid-attenuated_inversion_recovery)

### 3 Topological properties and reachability

We shall now expand our specification in order to illustrate the use of reachability formulas. First of all, we declare a bunch of constants, to simplify the specification by identifying the various coloured squares (which could be thought of as points of interest, or as actors willing to move in the maze), the corridors, the walls, and the exit.

```

let rgbcol(r,g,b) =
    (red(img) =. r) & (green(img) =. g) & (blue(img) =. b)

let corridor = rgbcol(255,255,255)
let exit = rgbcol(0,255,0)
let trapdoor = rgbcol(255,128,128)

let cyan = rgbcol(0,255,255)
let orange = rgbcol(255,128,0)
let magenta = rgbcol(255,0,255)
let blueSq = rgbcol(0,0,255)

let all = cyan | orange | magenta | blueSq

```

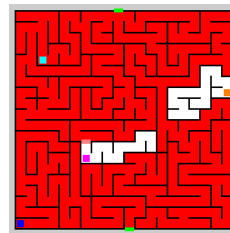
For reachability properties, we will employ the operator  $\sim>$ . A pixel  $p$  satisfies  $a \sim> b$  if there is a path starting in  $p$  and ending in a point satisfying  $b$ , such that all points of the path satisfy  $a$ <sup>9</sup>. Let us now find the pixels belonging to the corridors from which the exit can be reached. These are the pixels from which a path can be drawn traversing the corridors, until a pixel which is **adjacent** to the exit is found. The property of being adjacent to the exit can be expressed using the *near* operator, denoted by  $\mathbf{N}$  in *ImgQL*. Thus, the points from which an exit can be reached are represented by the expression `freeCorridor` below.

```

let freeCorridor = corridor  $\sim>$  ( $\mathbf{N}$  exit)

save "output/example04.png"
    view(freeCorridor)

```



Additionally, we can identify the points of interest from which an exit can be reached passing through corridors, by chaining two reachability properties<sup>10</sup>. Indeed, only the cyan and blue squares are coloured in red.

<sup>9</sup> This is a variant of the  $\rho$  operator used in [8], the difference being that with  $\rho$ , the extremes of the path do not need to satisfy  $a$ .

<sup>10</sup> The reader should now pause, and understand (even by experimenting) why actually, two reachability properties *are* needed.

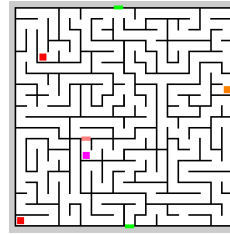


```

let freeSquares = all ~> N freeCorridor

save "output/example05.png"
      view(freeSquares)

```



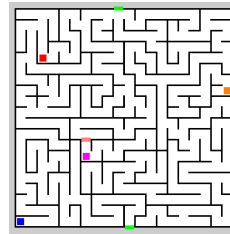
Now consider our maze as if it was an abstract model of an emergency scenario. A person, who could be in one of the cyan, blue, or orange spots holds the key to the pink trapdoor, and ought to go there, open the trapdoor and rescue a person in the magenta spot, then reach the exit. Let us try to express it using the  $\sim>$  operator. There could be more than one way of doing so. Our proposal is a chain of nested reachability properties. Indeed, only the cyan square satisfies the specification.

```

let rescuer = (cyan | blueSq | orange)
              ~> N (corridor ~>
                  (N (trapdoor ~>
                      (N (corridor ~>
                          (N (magenta ~>
                              (N ((corridor | trapdoor) ~>
                                  (N exit ))))))))
                  ))))

save "output/example06.png" view(rescuer)

```



## 4 Global properties and Region Calculi

VoxLogicA has a number of *global* operators, among which those that can compute the maximum and minimum of the values where a specific Boolean-value image is true, or that can compute the volume (number of pixels) of a Boolean-valued image. The results of such operators can be inspected using the **print** instruction, having a syntax similar to that of the **save** instruction.

For instance, the volume of the corridors (if useful for any purpose), the number of pixels of the whole image, and the ratio between the two, can be computed and displayed in the log as follows (below, `tt` is the Boolean operator “true”, which holds at any pixel):

```

print "corridors volume" volume(corridor)
print "image volume" volume(tt)
print "corridors / total volume" volume(corridor) ./ volume(tt)

```

```

[258ms] [user] image volume=1048576.0
[274ms] [user] corridors volume=786307.0
[275ms] [user] corridors / total volume=0.7498807907

```

Typically, such values are then collected by a script – for instance, when running VoxLogicA against a dataset as explained in Section 2.4 – and eventually used for statistical purposes. A simple application of the `volume` operator is to check whether, in a given image, there exists a point having a given property, by checking whether the volume of the pixels satisfying that property is greater than 0. For instance, in order to check whether there are any “rescuers” in an image of a maze, we can do as follows:

```
let exists(x) = volume(x) .>. 0
print "existsRescuer" exists(rescuer)
```

```
[545ms] [user] existsRescuer=true
```

By the above, VoxLogicA can also be used as a method to *query* datasets of images in order to identify those that satisfy specific requirements. For instance, a real-world scenario could be that of using the procedure for brain tumour segmentation described in [8] in a dataset of patients, in order to identify cases with particularly large brain tumours, or e.g., where the tumour is very close to the cerebellum. Similarly, the nevus segmentation procedure of [6] could be turned into a method to identify patients with nevi having specific characteristics (e.g. ratio between border and surface, etc.). The position paper [5] further elaborates on this idea.

Expanding on global operators, the paper [20] demonstrated that the classical binary operators of the family of *Region Calculi* can be defined in ImgQL. More precisely, given two regions, it is possible to check whether such regions are *disconnected*, *externally connected*, *equal*, *partially overlapping*, or if one is a *tangential* or *non-tangential proper part* of the other.

The operators of the region calculus have been implemented in a VoxLogicA library, consisting in the file `RegionCalculus.imgql` residing in the same directory as the VoxLogicA executable. Users can load such library by writing:

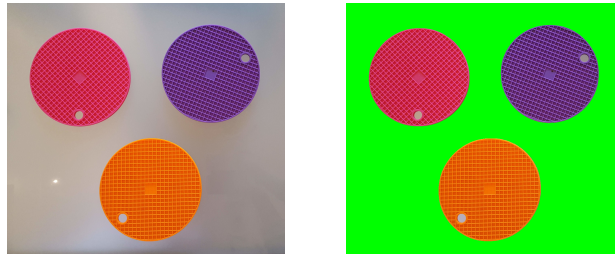
```
import "RegionCalculus.imgql"
```

Recall that also the standard library `"stdlib.imgql"`, which is automatically imported, resides in the same directory. It can be useful for the reader to inspect these two files, in order to learn about the pre-defined derived operators, and how they can be defined using the basic primitives of VoxLogicA.

## 5 Advanced topics: background removal, distance, filtering, texture similarity

In this section, we illustrate a slightly more advanced example, making use of reachability and the built-in *border* predicate to remove the background from a coloured image. Such method is actually used in [8] to remove the background from the dataset of brain images employed therein and identify the area containing the brain. In passing, we will illustrate a kind of “filter” pattern used to *smoothen* images in order to remove non-essential details, using the built-in *distance transform* operator.

We will use the image on the left below, depicting three coloured plastic discs laying on a grey surface; note that although the background is quite uniform, it is not just made of a single colour, due to illumination. Our example will be aimed at “masking” the background from the image, by colouring it in green, in order to obtain the image on the right. Note that the exercise does *not* require to colour in black the parts of the background that are inside the smaller holes of the three coloured discs, which therefore remain grey.

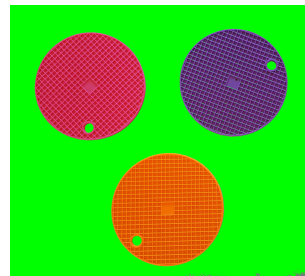


As a first step, the most obvious thing to do is to apply a threshold on the red, green and blue components of the image, in order to separate the grey areas, in which all the three components have a high value at the same time, from the coloured areas, where some components are predominant.

```
load i = "three_coloured_items.png"

let greyish =
  (red(i) >. 120) & (green(i) >. 120) &
  (blue(i) >. 120)

let view(x) = overlayColor(x,i,0,255,0)
save "output/greyish.png" view(greyish)
```



We have again defined a `view` function, this time showing our results in green in order to maximise contrast. Note that the threshold we have used works quite well, but still leaves some fuzzy margin near to the lower-right corner of the image (where the background is darker). Moreover, quite obviously, also the inner part of the holes in the coloured discs has been selected by the threshold. Finally, we note that there are small areas that are not captured by the threshold, mostly, close to the border of the discs, and the purple disc also contains some noise, due to some grey shadows in the picture. Such issues are clearly visible, for instance, by zooming in on the relevant areas, as done below.

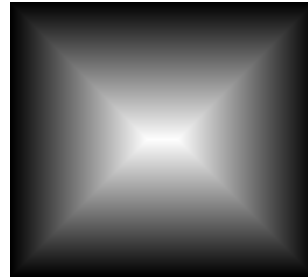


In order to exclude the “inner” points from our selection (inner parts of the holes and green points in the purple disc), the built-in predicate `border` of VoxLogicA can be of help. Such predicate is true only at the borders of an image. This means the area where `border` is true is only 1 pixel wide, and would not be clearly visible in a picture. So we have an excuse to illustrate the *distance transform* operator, since it can be used to thicken the border and visualise it more clearly. Later, we will use the same operator for *smoothing*.

The distance transform `pdt(x)` is an imaging primitive that, given a Boolean-valued image  $x$ , returns a number-valued image where each pixel  $p$  contains the numeric value of the *Euclidean distance* of  $p$  from the points where  $x$  is true. This is defined as the minimum of the distance of  $p$  from *any* point where  $x$  is true. To be more precise, below we use the so-called “positive” distance transform, which is zero on the points where  $x$  is true (hence the “p” in `pdt`). In image formats that have a notion of physical dimension of pixels, the distance is expressed in millimetres; otherwise, the distance unit corresponds to the width of a pixel.

```
let normalise(x, v) =
  (x /. max(x)) *. v

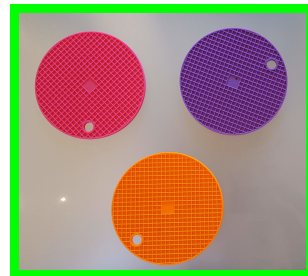
save "output/pdt.png"
  normalise(pdt(border), 255)
```



To ease visualization of the result, we defined the function `normalise(x, v)`, dividing the value of the image  $x$  in each pixel by the maximum value, so that the maximum in the result is 1; then by multiplying it by  $v$ , the maximum becomes  $v$ . When saving, we let  $v$  take the value 255 which is the maximum representable value in an 8-bit grayscale image<sup>11</sup>. Visually, the resulting image is dark in areas very close to the border, whereas pixels that are far from the border are coloured in more intense shades of white. By applying a threshold on the distance transform, we can visualize the border by “thickening it” as follows.

```
let thickBorder = pdt(border) <. 30

save "output/thickBorder.png"
  view(thickBorder)
```

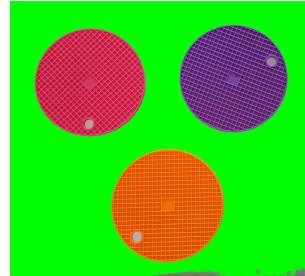


<sup>11</sup> In order to avoid issues related to overflow and low precision of 8-bit integers altogether, VoxLogicA can save images in the `nifti` format. Such format can use floating point values in pixels (and can also represent multi-dimensional images, for instance 3D MRI or CAT medical images). See <https://nifti.nimh.nih.gov/>.

Now that it is clear what the `border` predicate does, we can return to our background segmentation specification, and identify those greyish areas that touch the border, in order to separate them from the inside of holes and the noisy result of the threshold operation on the purple disc-

```
let greyishTouchBorder =
  greyish ~> border

save "output/greyishTouchBorder.png"
  view(greyishTouchBorder)
```

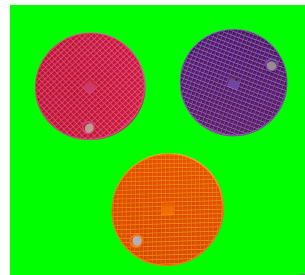


The resulting image is quite close to the result we have in mind, but not there yet. There is noise in the result, both on the lower-right corner of the image, and close to the border of the discs, as we already noted. In order to remove noise, very often in imaging some form of *smoothing* is used, as illustrated below.

```
let distgeq(x,y) = x .<= pdt(y)
let distleq(x,y) = x .>= pdt(y)
let flt(x,a) = distleq(x,distgeq(x,!a))
let dualSmoothen(x,a) =
  flt(x,a) | (!flt(x,!a))

let smooth =
  dualSmoothen(10,greyishTouchBorder)

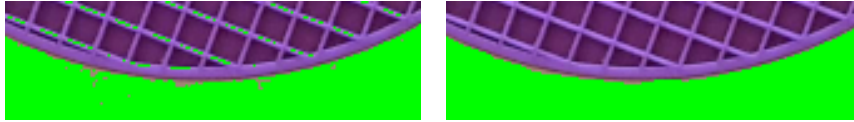
save "output/filtered.png" view(smooth)
```



We define the `flt(x,a)` function with two arguments, a number `x` and a Boolean-valued image `a`. The idea is that the area where `a` is true is first shrunk, by only keeping the points that lay at a distance greater or equal than `x` from its *complement* `!a`, and then enlarged by taking the points that lay at a distance less or equal than `x` from the “shrunk” image. The initial shrinking eliminates areas that are smaller in radius than `x`, whereas enlarging the result “fills” the resulting holes.

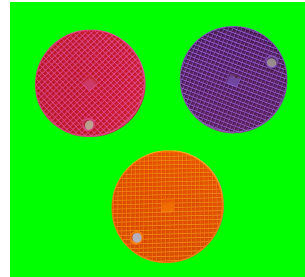
In previous work, the `flt` function is usually applied to a Boolean-valued image, in order to remove noise in the area where the image is *true*. In this example, however, both the part of `greyishTouchBorder` which has value *true* and that having value *false* may be noisy. Therefore, we define the `dualSmoothen` function that applies the `flt` function both to an image and to its complement, and combines the two results. To aid the intuition, the reader may think that the `dualSmoothen` function enlarges `flt(x,a)` by *adding* to it whatever point `p` that is *removed* from its complement `!a` in the expression `flt(x,!a)`. Technically, this is done by adding to it *any* point in `!flt(x,!a)` (which includes

both the points  $p$  in the above situation, and the points that are *already* in  $\text{flt}(x, a)$ . Zooming in demonstrates the combined effect of selecting only the part of `greyish` that touches the border, and of the dual smoothing. The result is shown on the right, against the image `greyish`, reported on the left, for comparison.



Finally, the remaining “holes” near to the border can be filled as follows.

```
let final = smooth | ((!smooth) ~> border)
save "output/final.png" view(final)
```



A very similar method has been successfully employed for 3D MRI-FLAIR dataset in the brain tumour case study of [8]. More advanced techniques can be used, involving, for instance, the *statistical texture similarity* operator, presented in [3,8]. The texture similarity operator associates to each pixel  $p$  a *similarity score* between  $-1$  and  $1$ , relating an area with a given radius  $r$  around  $p$  to a target region, denoted by a Boolean-valued image, by comparing the  $k$ -bins histograms of the area around  $p$ , and of the target region, using a method called *cross-correlation*. For instance, the background segmentation method employed in [6] finds the areas of the image that have a texture similar to the area close to the border. We refer the reader to the cited papers, and only show a similarity map obtained using such method, where darker areas are less similar to the area near to the border.

```
let similar(x, r, k) =
  crossCorrelation(r,
    intensity(i), intensity(i),
    x, min(intensity(i)),
    max(intensity(i)), k)

let simMap =
  similar((pdt(border) <. 3), 30, 4)

save "output/texture.png"
  normalise(simMap +. 1, 200)
```



The function `similar(x, r, k)` computes the similarity score with respect to a target region (Boolean valued image)  $x$ , with radius  $r$  and number of bins  $k$ . We report the full definition of `similar` for the reader to return to it, after studying the topic in more detail. We just note that we use a very low number of bins ( $k = 4$ ) and a high radius ( $r = 30$ ) in order to obtain a quite coarse-grained analysis, which yields a good similarity map for background segmentation purposes (the similarity map is meant to be thresholded just like we did with the red, green and blue components of the image in the beginning). Finally, observe that we normalise values between  $-1$  and  $1$ , therefore we add the value  $1$  to each pixel<sup>12</sup>.

## 6 Related work

As we already mentioned, the VoxLogicA approach stems from topological spatial logics. The reader interested in the theoretical developments behind this fascinating topic should consult the Handbook of Spatial Logics [1], containing several monographic chapters on selected topics in the field.

The development of SLCS in [16] has spawned a few research lines. The work in [32,33,4,31] proposes the *Signal Spatio-temporal Logic* (SSTL) that combines the analysis of continuous signals through the *Signal Temporal Logic* with the topological spatial operators of SLCS. In [34], SSTL has been used for specifying spatio-temporal patterns in the context of particle-based simulation, as part of a statistical spatio-temporal model-checking approach, following the method described in [18]. The results presented in [35] introduce a spatio-temporal logic for bigraphs, inspired by [24], and use the tool `topochecker` presented in [15] for verification. The recent work in [2] demonstrated that SLCS formulas can be interpreted in a fully distributed way, for monitoring purposes across a network. The research line started in [14] aims at providing a categorical generalisation of modal logics with reachability, based on *hyperdoctrines*, covering many examples such as fuzzy sets, algebraic structures, coalgebras, and also known cases such as Kripke frames and probabilistic frames. The study of model-based equivalences (such as bisimulation) and minimisation algorithms that are correct and complete with respect to logical equivalence of the Spatial Logic of Closure Spaces has been recently pursued in [21,22,27]. In [10], some of the authors of this paper co-authored an effort towards model checking of continuous space, by re-using the continuous semantics of SLCS given in [17] in the restricted, computation-friendly setting of models based on polyhedral valuations, which are *triangulated* to form *simplicial complexes*. As an application, *3D meshes* can be loaded and analysed using methods similar to those that have been illustrated in this tutorial. Also the recent work in [28] interprets SLCS on simplicial complexes; the focus therein is not on defining a notion of reachability which is compatible with the definition of [17]; rather, the authors exploit simplicial complexes as a description of relations between data, and the chosen accessibility relations between simplexes reflect such choice.

<sup>12</sup> We do not normalise the result to the maximum representable value 255, but just to 200, to make the lighter areas grey, which is more prominent on white paper.

Furthermore, without claiming to be exhaustive, we cite a few approaches to logic-based spatio-temporal analysis that are not directly related to SLCS. In [25] spiral electric waves—a precursor to atrial and ventricular fibrillation—are detected and specified using a spatial logic and model-checking tools. The formulas of the logic are learned from the spatial patterns under investigation and the onset of spiral waves is detected using bounded model checking. In the logical language SpaTeL [26], space is hierarchically divided in quadrants, and complex logic formulas, in the form of *quad-trees*, are built using machine learning methods. In [29], the authors define a logic language grounded on a chemical-based coordination model. Logic formulas are evaluated in a distributed manner by using an inference procedure which verifies them against the current global state of the system, checking whether the emergent global behaviour obeys to the required properties.

## 7 Outlook

We hope that reading this tutorial up-to here has not only initiated the reader to the basics of Spatial Model Checking and VoxLogicA, but has also raised some interest in the ongoing developments that will soon become relevant additions to the landscape of instruments devoted to spatial model checking. Currently, the VoxLogicA group is pursuing a few major research lines.

First and foremost, the immediate interest of the group is in advancing the healthcare related applications of Formal Methods and Spatial Model Checking in particular. Besides identifying new promising case studies, and improving the existing results, the integration, to some degree, of Machine Learning methods into logical specifications is an interesting scientific challenge. This could be used, for instance, to calibrate numeric parameters, or to accomplish some basic imaging tasks using Machine Learning, and coordinate them using explainable logical specifications to obtain more refined, complex results.

Very relevant for, but not limited to, the healthcare applications is the development of a dataset-oriented user interface that can leverage studies on the cognitive load on users (see e.g. [11]) in order to make logic-based analysis simpler to develop and more effective.

The natural setting in which such a user interface can be used is that of *interactive* development of logical specifications against training datasets. Currently, even if VoxLogicA is quite fast (often requiring no more than a few seconds to complete the analysis of a single case), running an analysis against a whole dataset is a *batch* (non-interactive) process. The progress on the implementation of spatial model checking on GPUs [12,13] may lead to a dramatic improvement in this respect.

Another way to reduce the computational cost of analysis is by making the models to be analysed *smaller*. The study of *minimization* algorithms up-to logical equivalence may be a key advancement in this direction [21,22].

Also relevant in so called “future healthcare” is the study of novel imaging modalities based on 3D meshes (instead of pixels / voxels); in the same vein, ar-



tificial vision and augmented reality applications are already starting to appear, especially in surgery. The work in [10] is the starting point of an effort in the direction of bringing the spatial analysis capabilities of VoxLogicA to the realm of 3D meshes. We note in passing that the applications of such methods are definitely not limited to the domain of healthcare, as 3D modelling is pervasive in several fields of modern Computer Science and its applications.

It is not difficult to imagine that a language such as ImgQL (dubbed a “query language” from its inception) could be useful as a true query language for datasets of images (think e.g. of the large radiological “Picture Archiving and Communication Systems (PACS)” that are nowadays in use in hospitals). One may be interested, for instance, in finding all the patients having a brain tumour of a particularly large size, or where the tumour may be too close to a specific organ at risk. In a recent position paper [5] some preliminary ideas are sketched in more detail.

Finally, we mention that, even though everything that was described in this paper is based on purely spatial analysis, the VoxLogicA group already has expertise in spatio-temporal modelling and logical specifications, through the tool `topochecker`, which was in a sense a predecessor to VoxLogicA but still has unique spatio-temporal verification capabilities (see [15,24,19,23,18]). Indeed, it is a planned future development to add such capabilities to VoxLogicA.

*Acknowledgements.* The authors wish to explicitly thank the current collaborators Nick Bezhanisvili, Giovanna Broccia, Laura Bussi, David Gabelaia, Fabio Gadducci, Gianluca Grilletti, Erik de Vink, and the coordinator of the *Formal Methods and Tools laboratory of ISTI-CNR* Maurice ter Beek, for their continuative support in turning the early developments of Spatial Model Checking, and the more recent work on VoxLogicA, into a solid research line, with several promising ongoing developments. The authors gratefully thank the Program Committee members of the *27th International SPIN Symposium on Model Checking of Software* (PC chairs Alfons Laarman and Ana Sokolova) for giving us the opportunity to disseminate our results to such an amazing audience.

## References

1. Aiello, M., Pratt-Hartmann, I., Benthem, van, J.: Handbook of Spatial Logics. Springer (2007). <https://doi.org/10.1007/978-1-4020-5587-4>
2. Audrito, G., Casadei, R., Damiani, F., Stolz, V., Viroli, M.: Adaptive distributed monitors of spatial properties for cyber-physical systems. *Journal of Systems and Software* **175**, 110908 (2021). <https://doi.org/https://doi.org/10.1016/j.jss.2021.110908>
3. 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). <https://doi.org/10.1007/s10009-019-00511-9>
4. Bartocci, E., Bortolussi, L., Loreti, M., Nenzi, L.: Monitoring mobile and spatially distributed cyber-physical systems. In: 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE. pp. 146–155. ACM (2017). <https://doi.org/10.1145/3127041.3127050>

5. Belmonte, G., Broccia, G., Bussi, L., Ciancia, V., Latella, D., Massink, M.: Querying medical imaging datasets using spatial logics (position paper). In: HEDA2021: The International Health Data Workshop 2021 in conjunction with 10th International Conference on Model and Data Engineering (MEDI 2021). Communications in Computer and Information Science, vol. To Appear. Springer (2021)
6. Belmonte, G., Broccia, G., Vincenzo, C., Latella, D., Massink, M.: Feasibility of spatial model checking for nevus segmentation. In: Proceedings of the 9th International Conference on Formal Methods in Software Engineering (FormalIESE'21). pp. 1–12. IEEE (2021). <https://doi.org/10.1109/FormalIESE52586.2021.00007>
7. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: Innovating medical image analysis via spatial logics. In: From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday. LNCS, vol. 11865, pp. 85–109. Springer (2019). [https://doi.org/10.1007/978-3-030-30985-5\\_7](https://doi.org/10.1007/978-3-030-30985-5_7)
8. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: Voxlogica: A spatial model checker for declarative image analysis. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS. LNCS, vol. 11427, pp. 281–298. Springer (2019). [https://doi.org/10.1007/978-3-030-17462-0\\_16](https://doi.org/10.1007/978-3-030-17462-0_16)
9. Benthem, van, J., Bezhanishvili, G.: Modal logics of space. In: Handbook of Spatial Logics [1], pp. 217–298. [https://doi.org/10.1007/978-1-4020-5587-4\\_5](https://doi.org/10.1007/978-1-4020-5587-4_5)
10. Bezhanishvili, N., Ciancia, V., Gabelaia, D., Grilletti, G., Latella, D., Massink, M.: Geometric model checking of continuous space (2021), <https://arxiv.org/abs/2105.06194>
11. Broccia, G., Milazzo, P., Ölveczky, P.C.: Formal modeling and analysis of safety-critical human multitasking. *Innov. Syst. Softw. Eng.* **15**(3-4), 169–190 (2019). <https://doi.org/10.1007/s11334-019-00333-7>
12. Bussi, L., Ciancia, V., Gadducci, F.: A spatial model checker in GPU (extended version). *CoRR* **abs/2010.07284** (2020), <https://arxiv.org/abs/2010.07284>
13. Bussi, L., Ciancia, V., Gadducci, F.: A spatial model checker in GPU. In: 41st International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE). LNCS, vol. To appear. Springer (2021)
14. Castelnovo, D., Miculan, M.: Closure hyperdoctrines, with paths. *CoRR* **abs/2007.04213** (2020), <https://arxiv.org/abs/2007.04213>
15. Ciancia, V., Grilletti, G., Latella, D., Loreti, M., Massink, M.: An experimental spatio-temporal model checker. In: Software Engineering and Formal Methods - SEFM 2015 Collocated Workshops. LNCS, vol. 9509, pp. 297–311. Springer (2015). [https://doi.org/10.1007/978-3-662-49224-6\\_24](https://doi.org/10.1007/978-3-662-49224-6_24)
16. Ciancia, V., Latella, D., Loreti, M., Massink, M.: Specifying and verifying properties of space. In: Theoretical Computer Science - 8th IFIP TC 1/WG 2.2 International Conference, TCS. LNCS, vol. 8705, pp. 222–235. Springer (2014). [https://doi.org/10.1007/978-3-662-44602-7\\_18](https://doi.org/10.1007/978-3-662-44602-7_18)
17. Ciancia, V., Latella, D., Loreti, M., Massink, M.: Model Checking Spatial Logics for Closure Spaces. *Logical Methods in Computer Science* **Volume 12, Issue 4** (Oct 2016). [https://doi.org/10.2168/LMCS-12\(4:2\)2016](https://doi.org/10.2168/LMCS-12(4:2)2016)
18. Ciancia, V., Latella, D., Massink, M., Paškauskas, R., Vandin, A.: A tool-chain for statistical spatio-temporal model checking of bike sharing systems. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA, Part I. LNCS, vol. 9952, pp. 657–673 (2016). [https://doi.org/10.1007/978-3-319-47166-2\\_46](https://doi.org/10.1007/978-3-319-47166-2_46)

19. Ciancia, V., Gilmore, S., Grilletti, G., Latella, D., Loretì, 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>
20. Ciancia, V., Latella, D., Massink, M.: Embedding RCC8D in the collective spatial logic CSLCS. In: *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*. LNCS, vol. 11665, pp. 260–277. Springer (2019). [https://doi.org/10.1007/978-3-030-21485-2\\_15](https://doi.org/10.1007/978-3-030-21485-2_15)
21. Ciancia, V., Latella, D., Massink, M., de Vink, E.: Towards spatial bisimilarity for closure models: Logical and coalgebraic characterisations. *CoRR* **abs/2005.05578** (2020), <https://arxiv.org/abs/2005.05578>
22. Ciancia, V., Latella, D., Massink, M., de Vink, E.: On bisimilarities for closure spaces - preliminary version (2021), <https://arxiv.org/abs/2105.06690>
23. Ciancia, V., Latella, D., Massink, M., Paškauskas, R.: Exploring spatio-temporal properties of bike-sharing systems. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops*. pp. 74–79. IEEE Computer Society (2015). <https://doi.org/10.1109/SASOW.2015.17>
24. Grilletti, G.: *Spatio-Temporal Model Checking: Explicit and Abstraction-Based Methods*. Master’s thesis, University of Pisa (2016), <https://etd.adm.unipi.it/t/etd-06282016-191103/>
25. Grosu, R., Smolka, S., 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/http://doi.acm.org/10.1145/1467247.1467271>
26. Haghghi, 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*. pp. 189–198. HSCC ’15, ACM (2015). <https://doi.org/10.1145/2728606.2728633>
27. Linker, S., Papacchini, F., Sevegnani, M.: Analysing spatial properties on neighbourhood spaces. In: *45th International Symposium on Mathematical Foundations of Computer Science, MFCS. LIPIcs*, vol. 170, pp. 66:1–66:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPIcs.MFCS.2020.66>
28. Loretì, M., Quadrini, M.: A spatial logic for a simplicial complex model (2021), <https://arxiv.org/abs/2105.08708>
29. Luca De Angelis, F., Di Marzo Serugendo, G.: A logic language for run time assessment of spatial properties in self-organizing systems. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. pp. 86–91 (2015). <https://doi.org/10.1109/SASOW.2015.19>
30. Menze, B.H.e.a.: The multimodal brain tumor image segmentation benchmark (brats). *IEEE Transactions on Medical Imaging* **34**(10), 1993–2024 (2015). <https://doi.org/10.1109/TMI.2014.2377694>
31. Nenzi, L., Bortolussi, L., Ciancia, V., Loretì, M., Massink, M.: Qualitative and Quantitative Monitoring of Spatio-Temporal Properties with SCTL. *Logical Methods in Computer Science* **14**(4), 1–38 (2018). [https://doi.org/10.23638/LMCS-14\(4:2\)2018](https://doi.org/10.23638/LMCS-14(4:2)2018)
32. Nenzi, L., Bortolussi, L.: Specifying and monitoring properties of stochastic spatio-temporal systems in signal temporal logic. In: Haviv, M., Knottenbelt, W.J., Maggi, L., Miorandi, D. (eds.) *8th International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS*. ICST (2014). <https://doi.org/10.4108/icst.valuetools.2014.258183>

33. Nenzi, L., Bortolussi, L., Ciancia, V., Loretì, M., Massink, M.: Qualitative and quantitative monitoring of spatio-temporal properties. In: Runtime Verification - 6th International Conference, RV. Lecture Notes in Computer Science, vol. 9333, pp. 21–37. Springer (2015). [https://doi.org/10.1007/978-3-319-23820-3\\_2](https://doi.org/10.1007/978-3-319-23820-3_2)
34. Ruschinski, A., Wolpers, A., Henning, P., Warnke, T., Haack, F., Uhrmacher, A.M.: Pragmatic logic-based spatio-temporal pattern checking in particle-based models. In: Winter Simulation Conference, WSC 2020. pp. 2245–2256. IEEE (2020). <https://doi.org/10.1109/WSC48552.2020.9383908>
35. Tsigkanos, C., Kehrer, T., Ghezzi, C.: Modeling and verification of evolving cyber-physical spaces. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017. pp. 38–48. ACM (2017). <https://doi.org/10.1145/3106237.3106299>