Università degli Studi di Pisa

DIPARTIMENTO DI MATEMATICA

Tesi di laurea magistrale

Spatio-Temporal Model Checking: Explicit and Abstraction-Based Methods

Candidato:
Gianluca Grilletti
Matricola 483633

Relatore:
Ugo Montanari

Correlatore:
Vincenzo Ciancia

Anno Accademico 2015–2016

Abstract

Model checking consists in specifying properties of a system with an appropriate logic and in verifying them using algorithms that are recursive over the syntax of such logic. Currently this method is widely applied to develop verification tools for both software and hardware debugging.

An interesting branch of this subject developed in the last years is that of *spatio-temporal model checking*, namely model checking applied to study properties of spatial systems or geometrical objects. For this purpose, specific logics are defined and studied, called *spatio-temporal logics*. An example of these logics is `STLCS` (*Spatio-Temporal Logic of Closure Spaces*), a logic whose semantics is based on a generalization of topological spaces. The author contributed in writing the algorithms for the model checking procedure for this logic and an actual implementation of the model checker.

In this thesis, we present recent research on a model checking procedure (currently implemented in an experimental tool) for `STLCS`. Furthermore, we study novel techniques and algorithms for its spatial sublogic `SLCS` (*Spatial Logic of Closure Spaces*) to improve the efficiency of the existing model checking procedures, by extending the approaches of symbolic model checking and `CEGAR` (counter example guided abstraction-refinement) to the spatial case.

Contents

Introduction

Model checking is a method for formally verifying finite-state concurrent systems, conceived by Edmund M. Clarke and E. Allen Emerson to solve the *Concurrent Program Verification* problem [1] and then further developed by several authors in the past years. This method proved to be incredibly efficient in finding flaws and logical errors in digital circuit designs, software, and communication protocols, and the two authors, together with their colleague Joseph Sifakis, were awarded the *Turing Award* (the so called *Nobel of computing*) for this novel technique. Currently this method is widely applied to develop verification tools for both software and hardware debugging.

The main idea behind this method is to describe a system by means of an appropriate logic and to develop *efficient* algorithms in order to test the truth of a formula (in case of truth-value semantics) or to find which entities in the model entail a formula (in case of a set-satisfaction semantics). In the literature, the most used logics are *temporal ones* such as `CTL` (Computation Tree Logic) and μ-calculus. These logics allow to describe several interesting properties of a model such as *reachability* and *safety*. Efficient verification algorithms for temporal logics have been designed throughout the last decades, and their optimization is still an active research topic.

As an example, the model checking procedure for CTL roughly proceeds as follows: by traversing the syntax tree of a formula φ from the leaves to the root, the procedure can recursively compute the semantics of each subformula, thus obtaining the semantics of φ. This is the case for `CTL` formulas, whose semantics is computable in linear-time over the structure of φ. For some logics this traverse has to be repeated several times, as in the case of μ-calculus. In this case the complexity of the procedure becomes *exponential* in the number of nested fixed-point operators, although maintaining a linear complexity in the size of the system.

Indeed, there are limitations on the size of models that can be considered treatable using this method, but since model checking is implemented to test hardware or to study configurations of a certain system, this procedure has to deal with really large systems, even with more than 10^{20} states [2]. So in some cases a linear-time complexity *is not enough*. Several techniques and heuristics have been developed to achieve a consistent speed up when considering particular classes of models. Two important examples are *symbolic*

model checking and *abstraction techniques*.

Symbolic model checking is a technique developed by McMillan [3] to study large systems, and it does so by eliminating "redundant information" in the description of subsets of the model using `OBDDs` (Ordered Binary Decision Diagrams). This method proves to be really efficient, especially when the model considered is obtained by a parallel product of systems, as in the original problem studied by Clarke and Emerson.

On the other hand, *abstraction techniques* work by building an *approximation* of the model called *abstraction* and by analyzing it to obtain information on *how* to perform the model checking procedure. This is the case of `CEGAR` (Counter Example Guided Abstraction Refinement), a procedure developed in 2003 and now the basis of a large number of abstraction techniques [4]. The main idea of `CEGAR` is to build an abstraction that is also a *simulation* of the original model, use it to compute the semantics of a formula and then, if the semantics of the formula cannot be clearly determined, adjust the abstraction obtaining a finer approximation of the original model.

Both those techniques are broadly applied and achieve a significant speed up when compared with the classic procedures. This shows us that developing *ad hoc* algorithms to solve specific problems is a winning strategy in model checking, and also that the priority when searching for an efficient algorithm is to *reduce the size of the model*.

In this thesis, we focus on relatively recent models and logics that describe *space*. A *spatial model* is a formal description of a geometrical entity (such as a topological space, a metric space, and so on) by means of a logic, that we will refer to with the evocative name *spatial logic*. As the concept of spatial model is so generic, several formalizations are considered in the literature, and for each of them several logics are studied. The `MCP` (*Model Checking Problem*) for these logics assumes a different flavor, as the algorithms now have to take into account the spatial structure of the model in order to be efficient.

An example of spatial logic currently under study, which will be analyzed in this document, is `SLCS`, the *Spatial Logic of Closure Spaces*. This is a modal logic created to describe properties of a system using *closure spaces*, a generalization of topological spaces [5]. The logic can express some fundamental spatial properties of a system, such as *reachability*, *safety* and *proximity*, while retaining decidability and linear-time complexity for the model checking procedure.

A generalization of both the concepts of spatial and temporal logic is that of *spatio-temporal logic*. The aim of a spatio-temporal logic is to formally describe the properties of a spatio-temporal model (i.e., the formal description of a spatial system that changes over time). These logics are usually much more expressive than the previous ones, as they allow to intertwine spatial and temporal properties. On the other hand, defining an *interesting* spatio-temporal logic for which the `MCP` is decidable and has a

low complexity is not easy.

In particular, in this work, we study algorithms and optimizations related both to `SLCS` and to its spatio-temporal generalization `STLCS` (*Spatio-Temporal Logics of Closure Spaces*). Part of this thesis is devoted to detailing the semantics and model-checking procedure for `STLCS`, presented in a paper co-authored by the author of this document [6].

Since efficiency is of primary importance to solve the `MCP`, it is necessary to find new algorithms to speed up the model checking procedure also in the spatial case. In this document we will present two novel algorithms to achieve this speed up for the logic `SLCS`, based on McMillan's symbolic model checking and on `CEGAR` respectively.

In particular, the new abstraction-refinement algorithm based on `CEGAR` tries to preserve the spatial structure of the model while constructing a *succession of gradually finer abstractions*. Since simulation relations don't usually preserve spatial properties (such as *proximity* and *connectivity*) we can't apply the same technique used in `CEGAR` to define and refine the abstraction. To solve this problem, a new concept of model is here introduced and studied, namely the *multi-focus model* (or `MFM` in short). The aim of this new mathematical object is to approximate the initial model by *identifying some spatial regions* in single *abstract states*. By defining an appropriate semantics for `MFMs`, a suitable *transfer theorem* can be achieved, that relates entailment in the abstraction with entailment in the original model, and this result can then be translated into an actual algorithm.

In Chapter 1 we introduce the basic theory of model checking and the algorithms to solve the `MCP` for `CTL` and μ-calculus. In Chapter 2 we present *symbolic model checking* and two abstraction methods, namely *`CEGAR`* and *modal `CEGAR`*, applied to solve the `MCP` for the logics `CTL` and μ-calculus. In Chapter 3 we briefly present spatial and spatio-temporal logics, and focus on `SLCS` and `STLCS`, presenting the algorithms developed in [15] and [6]. In Chapter 4 we adapt *symbolic model checking* to `SLCS` and present the new abstraction-refinement algorithm inspired by the `CEGAR` approach.

Chapter 1

Background

In this chapter we will introduce the basic notions to understand the results in the rest of the document. In particular we will focus on the *model checking problem* (abbreviated as `MCP`) for the logics `CTL` and μ-calculus.

The model checking problem is quite simple to define. Fix a logic $\mathcal{L}$ and a semantics $[\![\bullet]\!]$ for this logic. The `MCP` consists in finding an algorithm that, given a model $\mathcal{M}$ and a formula φ, *compute* the semantics of φ at $\mathcal{M}$.

So for example, if we consider the theory of groups, given a group G and a formula φ, the model checking problem asks to find the *truth value* of φ at G. In this example, the semantics is a *decisional semantics*, meaning that $[\![\varphi]\!] \in \{\top, \bot\}$ (in more generality, that $[\![\varphi]\!]$ lies in a fixed set of truth values).

We obtain a slightly more complex example if we take a *satisfaction-set semantics*, namely a semantics whose models consist of a domain with some additional structure and $[\![\varphi]\!]$ is a subset of the domain. So taking again groups, the model checking problem asks, given G a group and $\psi(x)$ a formula, to find *all the elements* of G that entail the formula.

These last examples show us that the model checking problem is *not always solvable*, as several problems arise: is the formula φ decidable? Is the satisfaction set of $\psi(x)$ enumerable? How do we deal with high cardinality models?

In this chapter we will study the `MCP` only for two specific cases, namely the logic `CTL` and the logic μ-calculus. These are modal logics developed to study properties of graphs, in particular of *paths in graphs*. Even if they are expressive, these logics are somewhat *tame*, in the sense that their associated model checking problem is *solvable in linear time* in the encoding of the graph. They are widely studied and are the basis of several interesting applications, such as software and hardware debugging and optimization, so solving their associated `MCP` is of primary interest.

1.1 CTL Model Checking

We fix here a set AP of *atomic propositions* (or simply *propositions*). For our purposes we don't need to make any hypothesis on the cardinality of AP.

We now introduce CTL, a modal logic widely used to describe properties of systems by representing them with appropriate graph-like structures.

Definition 1.1.1 (CTL syntax)**.** We define the syntax of CTL as

$$\begin{aligned} \langle\varphi\rangle &\models \bot \mid p \mid \langle\varphi\rangle\wedge\langle\varphi\rangle \mid \neg\langle\varphi\rangle \mid \mathtt{E}\langle\Phi\rangle \mid \mathtt{A}\langle\Phi\rangle \\ \langle\Phi\rangle &\models \mathtt{X}\langle\varphi\rangle \mid \mathtt{F}\langle\varphi\rangle \mid \mathtt{G}\langle\varphi\rangle \mid \langle\varphi\rangle\mathtt{U}\langle\varphi\rangle \end{aligned}$$

where p is a generic element of AP.

We will call formulas of the first sort *state formulas* and formulas of the second sort *path formulas*.

The intuition is that this logic describes properties of nodes and paths. As the names suggest, a state formula describe a property of nodes, so we will define what it means that "a node entails a state formula". In the same way, a path formula has to describe a property of a path in the graph, and so we will define what it means that "a path entails a path formula".

Let's begin by defining the objects we will use as models, the *Kripke frames*.

Definition 1.1.2 (Kripke Frame)**.** A *Kripke frame* (or KF) is a tuple $\mathcal{M} = \langle S, \to, L\rangle$ where

- $(S, \to)$ is a graph (so S is a set of states and $[\to] \subseteq S \times S$ is a binary relation) where $[\to]$ is a total relation (i.e., for every $s \in S$ there exists $t \in S$ such that $s \to t$).
- $L : \mathtt{AP} \times S \to \{\top, \bot\}$ is called the *labeling* of the KF.

With L_p we will indicate the function $L(p, \bullet)$, for $p \in \mathtt{AP}$. Moreover, we call a *path* of the KF an infinite sequence of $s_0, s_1, \cdots \in S$ such that $s_i \to s_{i+1}$.

Usually we will use the symbol π to indicate a path, and with π_i we will indicate its i-th state.

Moral 1.1.3. We can imagine a KF to be a graph with additional "local information". In particular, given an atomic proposition $p \in \mathtt{AP}$ we can ask if a node s of the graph "satisfies that atomic proposition". Formally, this corresponds to check if $L_p(s) = \top$.

Definition 1.1.4 (Semantics of CTL)**.** Given a KF $\mathcal{M}$ we want to define what it means that a state *entails* a state formula (in symbols $\mathcal{M}, s \vDash \varphi$)

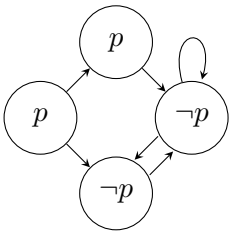


Figure 1.1: An example of KF. In each state we find the evaluation of the proposition p.

and a path *entails* a path formula (in symbols $\mathcal{M}, \pi \vDash \Phi$). So by induction over the structure of the formula we define:

$$
\begin{aligned}
&\mathcal{M}, s \nvDash \bot \\
&\mathcal{M}, s \vDash p \in \mathtt{AP} && \iff L_p(s) = \top \\
&\mathcal{M}, s \vDash \varphi_1 \wedge \varphi_2 && \iff \mathcal{M}, s \vDash \varphi_1 \text{ and } \mathcal{M}, s \vDash \varphi_2 \\
&\mathcal{M}, s \vDash \neg\varphi && \iff \mathcal{M}, s \nvDash \varphi \\
&\mathcal{M}, s \vDash \mathtt{E}\Phi && \iff \text{ exists a path } \pi \text{ such that } \pi_0 = s \text{ and } \mathcal{M}, \pi \vDash \Phi \\
&\mathcal{M}, s \vDash \mathtt{A}\Phi && \iff \text{ for all paths } \pi \text{ such that } \pi_0 = s \text{ it holds } \mathcal{M}, \pi \vDash \Phi
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{M}, \pi \vDash \mathtt{X}\varphi && \iff \mathcal{M}, \pi_1 \vDash \varphi \\
&\mathcal{M}, \pi \vDash \mathtt{F}\varphi && \iff \text{there exists } n \geq 0 \text{ such that } \mathcal{M}, \pi_n \vDash \varphi \\
&\mathcal{M}, \pi \vDash \mathtt{G}\varphi && \iff \text{for all } n \geq 0 \text{ it holds } \mathcal{M}, \pi_n \vDash \varphi \\
&\mathcal{M}, \pi \vDash \varphi_1 \mathtt{U} \varphi_2 && \iff \text{there exists } n \geq 0 \text{ such that } \mathcal{M}, \pi_n \vDash \varphi_2 \\
& && \qquad \text{and for all } k < n \ \mathcal{M}, \pi_k \vDash \varphi_1
\end{aligned}
$$

With $[\![\varphi]\!]^{\mathcal{M}}$ (respectively $[\![\Phi]\!]^{\mathcal{M}}$) we will indicate the set of states (paths) that entail the formula.

Moral 1.1.5. The semantics above intertwines properties of states (local properties) and properties of paths (global properties), and so it's quite expressive. To understand better the idea behind the various operators, we will now give them a name and a rapid description:

- $\mathtt{A}\Phi$ is the *"universal path quantifier"*. A simply tells us that the property Φ holds for all the paths starting at the given state; conversely, the *"existential path quantifier"* E tells that there exists at least one of such paths.

- $\mathtt{X}\varphi$ is the *"next operator"*. It tells us that the first state we reach by following the path given has the property φ. So for example $\mathtt{EX}\varphi$ means that there exists a direct successor of the state with the property φ.

- $\mathtt{F}\varphi$ is the *"eventually operator"*. It tells us that we will eventually reach a state with the property φ.
- $\mathtt{G}\varphi$ is the *"globally operator"*. It tells us that the property φ holds for *all* the states reached by the path.
- $\varphi_1 \mathtt{U} \varphi_2$ is the *"until operator"*. It states that, along the path, the property φ_1 holds *until* the property φ_2 doesn't become true, and also that a state with the property φ_2 will be reached.

Of course, some of the formulas, no matter the model, have the same semantics. For example, the formulas $\mathtt{AG}(p)$ ("all reachable states entail p") and $\neg\mathtt{EF}(\neg p)$ ("there is no reachable state entailing $\neg p$"). In particular, the so called *dual operators* are

- AF and EG ($[\![\mathtt{EF}p]\!] = [\![\neg\mathtt{AG}\neg p]\!]$)
- AG and EF ($[\![\mathtt{EG}p]\!] = [\![\neg\mathtt{AF}\neg p]\!]$)
- AX and EX ($[\![\mathtt{EX}p]\!] = [\![\neg\mathtt{AX}\neg p]\!]$)

In those cases we say that the formulas are *semantically equivalent* or simply *equivalent*.

Definition 1.1.6 (Semantic equivalence)**.** We say that two formulas of the same sort (meaning two state formulas or two path formulas) φ and ψ are *semantically equivalent* if

$$\text{For every KF } \mathcal{M}\text{: } [\![\varphi]\!]^{\mathcal{M}} = [\![\psi]\!]^{\mathcal{M}}$$

We will indicate with $\varphi \equiv \psi$ that the two formulas are equivalent.

The case of the U operator is a bit more complicated:

$$\mathtt{A}(p\mathtt{U}q) \equiv \neg\mathtt{E}(\neg q\mathtt{U}(\neg p \wedge \neg q)) \wedge \neg\mathtt{EG}\neg q$$

Using the equivalences above and $\mathtt{EF}p \equiv \mathtt{E}(\top\mathtt{U}p)$ is quite easy to show the following:

Definition 1.1.7 (Existential normal form)**.** A CTL formula is said to be in *existential normal form* (or ENF) if it is produced by the following grammar.

$$\langle\varphi\rangle \;\models\; \bot \mid p \in \mathtt{AP} \mid \langle\varphi\rangle\wedge\langle\varphi\rangle \mid \neg\langle\varphi\rangle \mid \mathtt{EX}\langle\varphi\rangle \mid \mathtt{EG}\langle\varphi\rangle \mid \mathtt{E}\langle\varphi\rangle\mathtt{U}\langle\varphi\rangle$$

Lemma 1.1.8. *Every CTL formula is equivalent to a ENF formula. Moreover, there is a linear-time algorithm to associate to a formula φ a formula $\varphi^{\exists}$ in normal form.*

Remark 1.1.9. Note that we never stated that the normal form *is unique*. For example:

$$\mathtt{EG}(p) \equiv p \wedge \mathtt{EX}(\mathtt{EG}(p))$$

The proof is quite trivial given the equivalences above, so we'll omit it. For a complete treatment of the topic see [8].

This result is quite useful, as now to solve the model checking problem for CTL we only need to give an algorithm to compute $[\![\varphi]\!]$ for φ a ENF formula, and this can be done by induction over the structure of the formula. More precisely, if we give an algorithm to compute the semantics of the formulas

$$\bot \mid p \mid p \wedge q \mid \neg p \mid \mathtt{EX}p \mid \mathtt{E}(p\mathtt{U}q) \mid \mathtt{EG}p \tag{1.1}$$

then we obtain an algorithm for all formulas, as for example we can compute $[\![\mathtt{EX}\varphi]\!]$ by considering $[\![\varphi]\!]$ as the interpretation of a phony predicate symbol p_φ and then applying the algorithm for the formula above.

Remark 1.1.10. Until now we never made hypothesis on the cardinality of the model. All we stated above is valid both for finite and infinite models, but now to find an algorithm to compute the semantics of an ENF formula we need the models to be finite.

We present now the algorithms to compute the semantics in the case of the formulas in (1.1). As the algorithm for the boolean operations is trivial, we present only the algorithms for the formulas $\mathtt{EX}p$, $\mathtt{E}(p\mathtt{U}q)$ and $\mathtt{EG}p$ (Algorithms 1, 2 and 3).

Notation 1.1.11. At the beginning of the algorithm pseudo-code, we indicate the data needed (in the field input) and the data we want to compute and return (in the field output).

With the notation var :=< *expression* >; we will indicate that we assign the value obtained evaluating < *expression* > to the variable var. < *expression* > could contain functions and operators already defined or mathematical expressions (to improve readability).

In some of the algorithms, we will suppose to have some primitive functions whose definition and implementation is clear from the context. As an example, in the algorithms here we make use of the function PredStates that, given a state s in input, gives back the set $\{t \in S | t \to s\}$.

Moral 1.1.12. The main ideas behind the algorithms are as follows:

- **$\mathtt{EX}p$:** You take the predecessors of the states that entail p, nothing more.
- **$\mathtt{EG}p$:** A state s entails $\mathtt{EG}p$ if and only if there is no path starting at s that reach a state for which $\neg p$ holds. So we can *overapproximate* the solution by taking $[\![p]\!]$, and then remove the states for which there is a "bad" path.

Algorithm 1: Algorithm to compute $\mathtt{EX}p$

input : $\mathcal{M} = \langle S, \rightarrow, L \rangle$, $\mathtt{EX}p$
output: $[\![\mathtt{EX}p]\!]^{\mathcal{M}}$

```
1 semP:= [[p]]^M;
2 res:= ∅;
3 for s ∈semP do
4 |   predSet:= PredStates(s);
5 |   res:= Union(predSet, res);
  return : res
```

Algorithm 2: Algorithm to compute $\mathtt{EG}p$

input : $\mathcal{M} = \langle S, \rightarrow, L \rangle$, $\mathtt{EG}p$
output: $[\![\mathtt{EG}p]\!]^{\mathcal{M}}$

```
1 res:= [[p]]^M;
2 semNotP:= S \ [[p]]^M;
3 corrosionLayer:= Intersection(predSet(semNotP), res);
4 while corrosionLayer ≠ ∅ do
5 |   res:= res \ corrosionLayer;
6 |   corrosionLayer:= Intersection(predSet(corrosionLayer), res);
  return : res
```

Algorithm 3: Algorithm to compute $\mathtt{E}(p\mathtt{U}q)$

input : $\mathcal{M} = \langle S, \rightarrow, L \rangle$, $\mathtt{E}(p\mathtt{U}q)$
output: $[\![\mathtt{E}(p\mathtt{U}q)]\!]^{\mathcal{M}}$

```
1 couldBe:= [[p]];
2 res:= [[q]]^M;
3 toAdd:= Intersection(predSet(res), couldBe);
4 couldBe:= couldBe \ toAdd;
5 while toAdd ≠ ∅ do
6 |   res:= Union(res, toAdd);
7 |   toAdd:= Intersection(predSet(toAdd), couldBe);
8 |   couldBe:= couldBe \ toAdd;
  return : res
```

- $\mathtt{E}(p\mathtt{U}q)$**:** This is quite similar to the previous case, but instead of over-approximating the solution, we *underapproximate* it by taking $[\![q]\!]$ and then by adding states that can reach the approximated solution with a "good" path.

Fact 1.1.13. *For finite* `KFs`*, the algorithms above terminate, are correct and their complexity is linear over the structure of the model and linear over the encoding of the formula.*

For countable `KFs`*, the algorithms above are correct and in ω steps give the solution.*

To be more precise: "the algorithm gives the solution in ω steps" means that, considering $\mathcal{P}(S) \cong 2^S$ with the usual product topology, the set `res` converges to $[\![\varphi]\!]$ (for φ the formula given in input to the algorithm).

So this solves the `MCP` for the logic `CTL`. Indeed, a linear time algorithm is the best that we can achieve as the semantics is a satisfaction-set one, so we achieved the fastest solution. But can we hope to be faster in the mean case? Is there any heuristic that let us solve the `MCP` *faster*, even for a particular class of instances? This problem will be examined in the next chapters.

1.2 μ-Calculus Model Checking

Another interesting logic to consider is the *μ-calculus*. If `CTL` describes the properties of infinite paths using the modal operators `EX`, `EU`, ..., μ-calculus does the same using *fixed-point operators*. In the following section we will describe the syntax and semantics of μ-calculus, we will study its links with `CTL` and then we will present algorithms to solve its associated `MCP`.

Definition 1.2.1 (Syntax of μ-calculus)**.** Fix an infinite number of variables `Var`. The syntax of the μ-calculus formulas is defined by the following grammar

$$\langle\varphi\rangle \models \bot \mid p \mid X \mid \neg\langle\varphi\rangle \mid \langle\varphi\rangle\wedge\langle\varphi\rangle \mid \Box\langle\varphi\rangle \mid \Diamond\langle\varphi\rangle \mid \mu X.\langle\varphi\rangle \mid \nu X.\langle\varphi\rangle$$

where $p \in \mathtt{AP}$ and $X \in \mathtt{Var}$.

We will refer to μ and ν as *quantifiers* in the following.

Fixed a formula, we say the occurrence of a variable X is *bound* if it occurs in a sub-formula of the type $\mu X.\varphi$ or $\nu X.\varphi$, otherwise we call it *free*. We say the variable X is bound if each of its occurrences is bound. We call a formula *closed* if all the variables that occur in it are bound.

We say a formula is *valid* if each bound occurrence of a variable occurs in the scope of an even number of negations. In the rest of the document with "μ-calculus formula" we will indicate a *valid formula* if not indicated differently.

Example 1.2.2. The formula $\mu X.\neg(\neg X \wedge Y)$ is valid as X occur under an even number of negations and Y is free.

On the other hand the formula $\nu Y.\mu X.\neg(\neg X \wedge Y)$ is not valid as Y now is bound and occurs under an odd number of negations.

Moral 1.2.3. The new ingredients we added are the following

- $\Box$ and $\Diamond$. Those quantifier-like operators stand for "for all successors it holds..." and "there exists a successor for which it holds..." respectively.

- μ and ν. Those are fixed-point operators and stand for "the smaller set such that..." and "the greatest set such that..." respectively.

Definition 1.2.4 (Environment)**.** Given $\mathcal{M} = \langle S, \to, L\rangle$ a `KF`, we call a function

$$g : \texttt{Var} \to \mathcal{P}(S)$$

an environment for $\mathcal{M}$.

With $g[X \mapsto A]$ we indicate the environment that coincide with g except at X, where it has value A. Formally:

$$g[X \mapsto A](Y) = \begin{cases} g(Y) & \text{if } Y \neq X \\ A & \text{if } Y = X \end{cases}$$

Definition 1.2.5 (μ-Calculus Semantics)**.** Given a `KF` $\mathcal{M}$ we want to define the semantic interpretation of a μ-calculus formula φ as the set of states that "entail" φ. As φ could contain free variables, we need to fix an environment g to define such interpretation.

To improve readability, we will define the function

$$\begin{aligned} Sem^{\mathcal{M}}_{\varphi,X,g} : \mathcal{P}(S) &\to \mathcal{P}(S) \\ A &\mapsto [\![\varphi]\!]^{\mathcal{M}}_{g[X\mapsto A]} \end{aligned}$$

that assigns to the subset $A \subseteq S$ the semantics of φ after changing the environment from g to $g[X \mapsto A]$.

So, fixed $\mathcal{M}$ and g an environment for $\mathcal{M}$, we define the semantics in-

terpretation $[\![\varphi]\!]_g^{\mathcal{M}}$ by the following inductive clauses:

$$\begin{aligned}
[\![\bot]\!]_g^{\mathcal{M}} &= \emptyset \\
[\![p]\!]_g^{\mathcal{M}} &= L_p^{-1}(\top) \\
[\![X]\!]_g^{\mathcal{M}} &= g(X) \\
[\![\neg\varphi]\!]_g^{\mathcal{M}} &= S \setminus [\![\varphi]\!]_g^{\mathcal{M}} \\
[\![\varphi \wedge \psi]\!]_g^{\mathcal{M}} &= [\![\varphi]\!]_g^{\mathcal{M}} \cap [\![\varphi]\!]_g^{\mathcal{M}} \\
[\![\Box\,\varphi]\!]_g^{\mathcal{M}} &= \{s \in S | \text{for all } t \text{ such that } s \to t\text{: } t \in [\![\varphi]\!]_g^{\mathcal{M}}\} \\
[\![\Diamond\varphi]\!]_g^{\mathcal{M}} &= \{s \in S | \text{there exists } t \text{ such that } s \to t \text{ and } t \in [\![\varphi]\!]_g^{\mathcal{M}}\} \\
[\![\mu X.\varphi]\!]_g^{\mathcal{M}} &= \text{lfp}\left(Sem^{\mathcal{M}}_{\varphi,X,g}\right) \\
[\![\nu X.\varphi]\!]_g^{\mathcal{M}} &= \text{gfp}\left(Sem^{\mathcal{M}}_{\varphi,X,g}\right)
\end{aligned}$$

where A is a generic subset of S and lfp and gfp stands for the least-fixed-point and the greatest-fixed-point operator respectively (considering the order relation $\subseteq$ over $\mathcal{P}(S)$).

Remark 1.2.6. Note that with the semantics above there is a certain redundancy, as it's easy to show that:

$$[\![\Diamond\varphi]\!] = [\![\neg\,\Box(\neg\varphi)]\!]$$
$$[\![\nu X.\varphi]\!] = [\![\neg\mu X.(\neg\varphi)]\!]$$

with this remark in mind, we can treat the $\Diamond$ and ν operators as shorthands.

Remark 1.2.7. Of course, the inductive definition above presuppose that *there is* a least and greatest fixed point for the function $Sem^{\mathcal{M}}_{\varphi,X,g}$, so the semantics could be not well-defined. To solve the problem the idea is to show that $Sem^{\mathcal{M}}_{\varphi,X,g}$ is *monotonic* whenever the formula φ is *valid*.

Lemma 1.2.8. *Fix a* KF $\mathcal{M}$ *and an environment* g*. Given a valid formula* φ*, the function* $Sem^{\mathcal{M}}_{\varphi,X,g}$ *is increasing monotonic.*

As a direct consequence of this lemma, by the Tarski-Knaster theorem we have the following:

Corollary 1.2.9. *The least-fixed-point and greatest-fixed-point of the function* $\left(A \mapsto [\![\varphi]\!]^{\mathcal{M}}_{g[X\mapsto A]}\right)$ *are both well-defined.*

The key lemma to prove these results (and the Tarski-Knaster theorem as well) is the following:

Lemma 1.2.10. *The least-fixed point of an increasing monotonic function* f *over* $\mathcal{P}(S)$ *can be computed as the limit of the trans-finite succession of subsets below:*

$$\begin{cases} f^{(0)} = \emptyset \\ f^{(\alpha+1)} = f(f^{(\alpha)}) \\ f^{(\lambda)} = \bigcup_{\alpha<\lambda} f^{(\alpha)} & \text{for } \lambda \text{ a limit ordinal} \end{cases}$$

So now that we showed that the semantics is well-defined, we can try to solve the MCP for the μ-calculus. As a matter of fact, given a finite model $\mathcal{M}$, computing the semantics of φ by induction over the structure of φ is quite trivial. The trickiest case is computing the semantics of the formulas of the type $\mu X.\psi$ and $\nu X.\psi$, but the lemma (1.2.10) gives us an effective procedure to do so in the finite case.

So using the results above, we have the following:

Lemma 1.2.11. *The MCP for μ-calculus and for the class of finite KF has a solution. Moreover, given $\mathcal{M}$ a KF and φ a formula, we can find an algorithm with complexity:*

- linear *over the size of $\mathcal{M}$;*
- exponential *over the number of nested quantifiers in φ.*

We omit here the pseudo-code (which is easy to obtain from the results above) and refer to [8].

Now we want to study the bonds between the two logics CTL and μ-calculus. As it's quite easy to obtain, μ-calculus can "simulate" some of the operators of CTL. Formally:

$$\text{For all } \mathcal{M} \text{ and } p \in \mathtt{AP}\text{: } [\![\mathtt{EX}p]\!]^{\mathtt{CTL}}_{\mathcal{M}} = [\![\Diamond p]\!]^{\mu}_{\mathcal{M}}$$
$$\text{For all } \mathcal{M} \text{ and } p \in \mathtt{AP}\text{: } [\![\mathtt{AX}p]\!]^{\mathtt{CTL}}_{\mathcal{M}} = [\![\Box p]\!]^{\mu}_{\mathcal{M}}$$

where we indicated with $[\![\bullet]\!]^{\mathtt{CTL}}$ and $[\![\bullet]\!]^{\mu}$ the semantics of CTL and μ-calculus respectively. Notice that we omitted the environment, as the semantics of closed formulas don't depend on it.

What is quite surprising, is that *every formula of CTL* can be "simulated" by a μ-calculus formula. Formally:

Lemma 1.2.12. *Given φ a CTL formula, there exists a closed μ-calculus formula φ^{μ} such that, for each model $\mathcal{M}$ it holds:*

$$[\![\varphi]\!]^{CTL}_{\mathcal{M}} \equiv [\![\varphi^{\mu}]\!]^{\mu}_{\mathcal{M}}$$

So we showed now that the μ-calculus is as expressive as CTL, but we can show that it is actually *more* expressive than CTL.

Fact 1.2.13. *The formula*

$$\nu X.\,(p \wedge \Diamond\Diamond X)$$

is not expressible in CTL (i.e., it doesn't exist a CTL formula equivalent to the above formula).

1.3 Complexity results

To end this chapter, we present some complexity results about the model checking problem and the satisfiability problem for CTL and μ-calculus. First of all, what is the satisfiability problem?

Definition 1.3.1 (Satisfiability problem)**.** Given a logic $\mathcal{L}$ with a fixed semantics, the satisfiability problem for the logic asks, given a formula φ, if the formula is entailed by *all the models* of the logic. A formula with this property is called a *tautology*.

Example 1.3.2. The satisfiability problem for μ-calculus asks, given a formula φ, if it's true for every state of every KF, for every possible environment. In symbols, if:

$$\text{For all } \mathcal{M} = \langle S, \to, L\rangle, \text{ for all } g : \mathtt{Var} \to \mathcal{P}(S):\ [\![\varphi]\!]^{\mathcal{M}}_g = S$$

A simple example of tautology is $p \vee \neg p$. A trickier example is:

$$(\mu X.\, (p \vee \Diamond X)) \vee (\nu X.\, (\neg p \wedge \Box\, X))$$

Morally, this formula states "I can reach a state at which p holds or every state I can reach entails $\neg p$".

Fact 1.3.3. *The following results hold:*

- *The model checking problem for CTL on finite models has* linear time complexity*.*
- *The satisfiability problem for CTL has* exponential time complexity*. Moreover, is EXP-complete ([9]).*
- *The model checking problem for μ-calculus is in NP$\cap$co-NP, but has* linear time complexity *if we consider only formulas with bounded alternation-depth complexity ([10]).*
- *The satisfiability problem for μ-calculus is in EXP ([11]).*

These results give us the limit of *how much* we can improve our algorithms and will help us study the expressive power of these logics.

Chapter 2

Heuristics

In this chapter we will focus on the main problem model checking faces: *efficiency*. In practice the flaw of the standard model checking approaches is that to represent a system a *huge* amount of states is needed, usually exponential compared to the description of the system.

A simple example can make this point really clear: given n memory cells in a computer memory, we can represent a *program* by a transition system where the states are strings of length n of zeros and ones, and transitions represent the effects on the memory of running the program. So, given n and the program as input, we have to deal with a system with 2^n states! This phenomenon is called *combinatorial explosion*, and the reason is quite clear from the example.

From a computation-time point of view, this is a disaster: even problems with a small description are not manageable with linear algorithms like the ones introduced in the previous chapter. To solve this problem several techniques have been developed and we focus here on two of them: *symbolic model checking* and *abstraction-refinement*.

2.1 Symbolic Model Checking

In 1992, McMillan introduced a new technique to solve the combinatorial explosion problem. When used to solve a combinatorial problem or when generating a description corresponding to several parallel systems, the number of states is exponential in the encoding of the problem. The idea of McMillan was to solve the problem *without* generating the system, but dealing only with an opportune encoding of it. This was the starting point of several model checking techniques later developed.

We now sketch the main idea of McMillan and the tools used to solve the problem in the `CTL` case.

Suppose the state-space is a set of the form $\{0, \ldots, 2^n - 1\}$ (we can always imagine to add states to round the number to a power of 2 and encode them

with "01" strings). Then:

- for each $p \in \texttt{AP}$ we can represent the function L_p as a *switching function* (i.e., a function $L_p : \{0,1\}^n \to \{0,1\}$).
- the adjacency relation can be represented as a switching function

$$\Delta : \{0,1\}^{2n} \to \{0,1\}$$
$$\langle s,t\rangle \mapsto \begin{cases} 1 & \text{if } s \to t \\ 0 & \text{otherwise} \end{cases}$$

  where we indicate with $\langle s,t\rangle$ the binary string obtained by joining the representation of s and t.

Switching functions are really useful to represent sets and set operations. For example:

- We can use switching functions to represent characteristic functions of sets.
- For the intersection of two sets represented by f and g respectively, we can use the function

$$f \wedge g : \{0,1\}^n \to \{0,1\}$$
$$s \mapsto f(s) \wedge g(s)$$

- To represent the *forward neighborhood* of a state s, i.e. the set

$$\mathrm{FN}(s) = \{t | s \to t\}$$

  we can use the switching function

$$\overline{x} \mapsto \Delta(s, \overline{x})$$

- To represent the *projection* of a function $h : \{0,1\}^{2n} \to \{0,1\}$ over the second n coordinates, we can use the function:

$$\exists \overline{x}.h(\overline{x}, \overline{x}') : \{0,1\}^n \to \{0,1\}$$
$$t \mapsto \begin{cases} 1 & \text{if there exists } s \text{ such that } h(s,t) = 1 \\ 0 & \text{otherwise} \end{cases}$$

We can even present the model checking procedure for `CTL` in terms of switching functions. Here we use the notation $g := \langle \textsf{expression} \rangle$ to indicate that we define a new switching function g, computed by expanding its definition. The expressions $\wedge$, $\vee$, $\exists \overline{x}'$ are treated primitives, as they can be implemented efficiently with adequate data structures. The symbolic algorithms are 4, 5 and 6.

Note that to make this computation efficient we need, given $f, g : 2^n \to 2$ and $h : 2^{2n} \to 2$, to compute efficiently the operations:

Algorithm 4: Algorithm to compute $\texttt{EX}p$

input : f_p, the switching function corresponding to $[\![p]\!]$
output: $f_{\texttt{EX}p}$, the switching function corresponding to $[\![\texttt{EX}p]\!]$

```
1 g:= ∃x̄′. (Δ(x̄, x̄′) ∧ f_p(x̄′));
2 return g;
```

Algorithm 5: Algorithm to compute $\texttt{EG}p$

input : f_p, the switching function corresponding to $[\![p]\!]$
output: $f_{\texttt{EG}p}$, the switching function corresponding to $[\![\texttt{EG}p]\!]$

```
1 f_0 := f_p;
2 j:= 0;
3 repeat
4 |   f_{j+1}(x̄) := f_j(x̄) ∧ ∃x̄′. (Δ(x̄, x̄′) ∧ f_j(x̄′));
5 |   j:= j +1;
6 until f_j == f_{j−1};
7 return f_j ;
```

Algorithm 6: Algorithm to compute $\texttt{E}(p\texttt{U}q)$

input : f_p, the switching function corresponding to $[\![p]\!]$
input : f_q, the switching function corresponding to $[\![q]\!]$
output: $f_{\texttt{E}(p\texttt{U}q)}$, the switching function corresponding to $[\![\texttt{E}(p\texttt{U}q)]\!]$

```
1 f_0 := f_q;
2 j:= 0;
3 repeat
4 |   f_j(x̄) := f_j(x̄) ∨ (f_p(x̄) ∧ ∃x̄′. (Δ(x̄, x̄′) ∧ f_j(x̄′)));
5 |   j:= j +1;
6 until f_j == f_{j−1};
7 return f_j ;
```

- $f \wedge g$
- $\neg f$ (as it is needed to compute the negation of a formula)
- $f \vee g$
- $\exists \overline{x}.h(\overline{x}, \overline{x}')$

Why are we interested in this representation? The general idea is that:

- a switching function is *easy to compress* using *ordered binary decision diagrams* (or `OBDDs` in short).
- computing boolean functions with `OBDDs` is really efficient.
- we can execute efficiently the `CTL` model checking algorithm using directly the `OBDD` representation of the functions.

Definition 2.1.1 (Ordered Binary Decision Diagram (`OBDD`))**.** Fix an ordered set of variables $\texttt{VarSet} = \{x_0 \prec x_1 \prec x_2 \prec \dots\}$. An `OBDD` is a tuple

$$\langle V = V_I \sqcup V_T, \texttt{succ}_0, \texttt{succ}_1, \texttt{var}, \texttt{val}, v_0 \rangle$$

where:

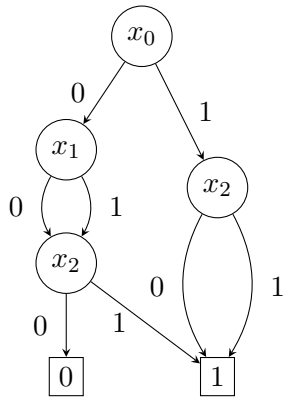
- V is a *finite* set of nodes divided in *internal nodes* (V_I) and *terminal nodes* (V_T).
- $\texttt{var} : V_I \to \texttt{VarSet}$ is a function that assign to each inner node a variable.
- $v_0 \in V$.
- $\texttt{succ}_0, \texttt{succ}_1 : V_I \to V$ are functions such that

$$\begin{gathered} v_0 \notin \texttt{succ}_0[V_I] \cup \texttt{succ}_1[V_I] \\ \text{For all } v \in V \setminus \{v_0\}\text{: } v \in \texttt{succ}_0[V_I] \cup \texttt{succ}_1[V_I] \\ \texttt{succ}_0(x) \in V_T \text{ or } \texttt{var}(x) \prec \texttt{var}(\texttt{succ}_0(x)) \\ \texttt{succ}_1(x) \in V_T \text{ or } \texttt{var}(x) \prec \texttt{var}(\texttt{succ}_1(x)) \end{gathered}$$

- $\texttt{val} : V_T \to \{0, 1\}$.

Basically we are giving a `DAG`-like structure to the set V where v_0 is the root, nodes of V_T are the leaves and the adjacency relation is given by the relation $\texttt{succ}_0 \cup \texttt{succ}_1$.

Example 2.1.2. In the figure an example of OBDD over the set of variables $\{x_0, x_1, x_2\}$.



Remark 2.1.3. Note that there are only two possible OBDDs where $v_0 \in V_T$, namely: $\boxed{0}$ and $\boxed{1}$. We will call them *trivial OBDDs*.

We now present some interesting properties of OBDDs.

Lemma 2.1.4. *Let $\mathfrak{B}$ be an OBDD over the variables*

$$\mathit{VarSet} = \{x_0 \prec x_1 \prec \dots\}$$

and w an internal node of $\mathfrak{B}$. Then define

$$\mathfrak{B}_w = \langle V_w, (\mathit{succ}_0)_w, (\mathit{succ}_1)_w, \mathit{VarSet}_w, \mathit{var}_w, w\rangle$$

where:

- *$V_w \subseteq V$ is the set of nodes reachable from w.*
- *$(\mathit{succ}_0)_w$, $(\mathit{succ}_1)_w$, var_w, val_w are the restriction of succ_0, succ_1, var, val to V_w.*

then $\mathfrak{B}_w$ is an OBDD over the set of variables $\{x \in \mathit{VarSet} \mid \mathit{var}(w) \preceq x\}$.

Thanks to this result, we can associate to an OBDD a switching function in a really intuitive way:

Definition 2.1.5 (Semantics of OBDDs)**.** Let $\mathfrak{B}$ be an OBDD over the set of variables $\{x_1 \prec \dots \prec x_n\}$, where we suppose $\mathtt{var}(v_0) = x_1$. Then we can associate to $\mathfrak{B}$ the switching function:

$$f_{\mathfrak{B}} = \begin{cases} 0 & \text{if } \mathfrak{B} = \boxed{0} \\ 1 & \text{if } \mathfrak{B} = \boxed{1} \\ (x_1 \wedge f_{\mathtt{succ}_0(v_0)}) \vee (\neg x_1 \wedge f_{\mathtt{succ}_1(v_0)}) & \text{otherwise} \end{cases}$$

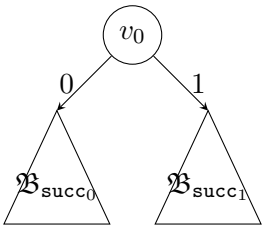
where $f_{\mathtt{succ}_0(v_0)}$ and $f_{\mathtt{succ}_1(v_0)}$ indicate the switching function associated to the OBDDs $\mathfrak{B}_{\mathtt{succ}_0(v_0)}$ and $\mathfrak{B}_{\mathtt{succ}_1(v_0)}$ respectively.

Remark 2.1.6. Note that the definition above is a recursive definition where the base cases are trivial OBDDs. In particular, the switching function is well-defined because the number of variables that appear in $\mathfrak{B}_{\mathtt{succ}_0(v_0)}$ and $\mathfrak{B}_{\mathtt{succ}_1(v_0)}$ is strictly less than the number of variables that appear in $\mathfrak{B}$.

Lemma 2.1.7. *Given $\mathfrak{B}$ a non trivial OBDD, its switching function is determined by the tuple*

$$\langle \mathtt{var}(v_0), f_{succ_0(v_0)}, f_{succ_1(v_0)} \rangle$$

Moreover, consider the OBDD $\mathfrak{B}'$ constructed by taking $\mathfrak{B}_{succ_0}$ and $\mathfrak{B}_{succ_1}$ as the 0-successor and the 1-successor of a common root respectively (as in the figure).



Then $f_{\mathfrak{B}} = f_{\mathfrak{B}'}$.

Notation 2.1.8. We will call two OBDDs $\mathfrak{B}$ and $\mathfrak{B}'$ *equivalents* if $f_{\mathfrak{B}} = f_{\mathfrak{B}'}$.

Moreover, we will indicate an OBDD constructed as in the lemma above with the notation $\mathfrak{B}_{\langle \mathtt{var}(v_0), \mathfrak{B}, \mathfrak{B}' \rangle}$.

It's also quite trivial to show that given a switching function we can define an OBDD that represents it.

Definition 2.1.9 (cofactors)**.** Let $f(x_1, \ldots, x_n)$ be a switching function. We call *negative cofactor* relative to the variable x_i the function

$$f|_{x_i=0}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

In the same way, we call *positive cofactor* relative to the variable x_i the function

$$f|_{x_i=1}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

Lemma 2.1.10. *Given a switching function $f(x_1, \ldots, x_n)$, consider the OBDD $\mathfrak{B}_f$ defined inductively as*

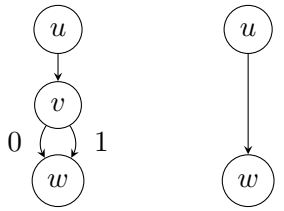
$$\mathfrak{B}_f = \begin{cases} \boxed{0} & \text{if } f = 0 \\ \boxed{1} & \text{if } f = 1 \\ \mathfrak{B}_{\langle x_1, \mathfrak{B}_g, \mathfrak{B}_h \rangle} & \text{otherwise} \end{cases}$$

where $g = f|_{x_1=0}$ and $h = f|_{x_1=0}$.

Then $f_{\mathfrak{B}_f} = f$.

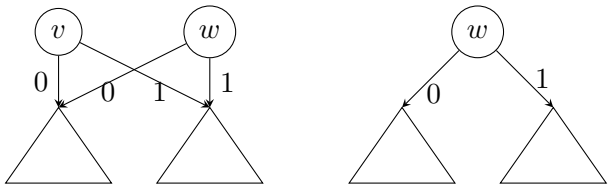
So we have seen that every OBDD represents a switching function, and that every switching function is represented by some OBDD. A question that arises quite naturally is if there is a way to make this correspondence "unique", or in other terms, if we can find a canonical OBDD representation of a switching function. For example, by applying the following reduction rules (if possible) to an OBDD already defined, is quite easy to see that we obtain an equivalent one.

- **Elimination rule:** If $v \in V_I$ and $\mathtt{succ}_0(v) = \mathtt{succ}_1(v) = w$, than eliminate v and redirect all incoming edges $u \to v$ to w.



- **Isomorphism rule:** Let $v \neq w$ be nodes of $\mathfrak{B}$ such that one of the followings hold:
  - $v, w \in V_T$ and $\mathtt{val}(v) = \mathtt{val}(w)$.
  - $v, w \in V_I$ and $\langle \mathtt{Var}(v), \mathtt{succ}_0(v), \mathtt{succ}_1(v) \rangle = \langle \mathtt{Var}(w), \mathtt{succ}_0(w), \mathtt{succ}_1(w) \rangle$.

  Then eliminate v and redirect all edges $u \to v$ to w.



Remark 2.1.11. Note that both operations eliminate a node and the rules can't be applied to trivial OBDDs. So as a result we obtain that each OBDD is equivalent to one which we can't reduce further.

In the following, we will show there exists a *normal form* for OBDDs, and in particular it is computable efficiently. First of all, let's define a special case of OBDD, one for which the information about the correspondent switching function cannot be further "compressed".

Definition 2.1.12 (Reduced OBDD)**.** We define a reduced OBDD (or ROBDD in short) as an OBDD $\mathfrak{B}$ such that for $v, w \in V$ it holds:

$$v \neq w \Rightarrow f_v \neq f_w$$

where f_v and f_w are the switching functions corresponding to $f_{\mathfrak{B}_v}$ and $f_{\mathfrak{B}_w}$ respectively.

Lemma 2.1.13 (ROBDDs equivalence classes)**.** *Two ROBDDs are equivalent* if and only if *are equal upon renaming of the nodes. Equivalently, equivalence and isomorphism coincide for ROBDDs.*

This is quite useful, as now to check if two ROBDDs are equivalent we simply need to check if they are equal (upon renaming), and this can be done in linear time.

A result quite trivial to show is that we can't apply reduction rules to ROBDDs. What is more interesting, is that also the converse holds! And so we have the following important results:

Lemma 2.1.14. *An OBDD is reduced* if and only if *it is not reducible using the rules above.*

Lemma 2.1.15 (OBDD normal form)**.** *Every OBDD is equivalent to* exactly one *ROBDD (upon renaming of the nodes). Moreover, the equivalent ROBDD can be obtained in* linear time *in the encoding of the initial OBDD.*

Corollary 2.1.16. *An OBDD* $\mathfrak{B}$ *is reduced* if and only if *is of minimal size in its equivalence class. Formally, if for every other equivalent* $\mathfrak{C}$ *it holds*

$$|V_{\mathfrak{B}}| \leq |V_{\mathfrak{C}}|$$

Remark 2.1.17. This result shows that ROBDDs are the smaller OBDDs representing a certain switching function. However, note that we fixed a variable ordering at the start of the chapter; to change the variable order means to change the size of the corresponding ROBDD. More about this topic can be found in [8].

Corollary 2.1.18. *Deciding the equivalence of two OBDDs is a linear time problem.*

To summarize the results above, we found a data structure such that:

- It represents switching functions.
- Can compress data in a canonical way (*reduction*).
- Equivalence is computable in linear-time in the OBDDs size.

To conclude we need to show that the CTL model checking operations can be carried using this structure. Formally, that given ROBDDs $\mathfrak{B}_f$ and $\mathfrak{B}_g$ (representing f and g respectively), we can compute efficiently $\mathfrak{B}_{f \wedge g}$, $\mathfrak{B}_{f \vee g}$, $\mathfrak{B}_{\neg f}$ and $\mathfrak{B}_{\exists \overline{x}.f}$ of bounded size, as they can be later reduced with the rules introduced above.

All these algorithms are presented and thoroughly described in [8].

2.2 Abstraction-Refinement for CTL

The main idea behind symbolic model checking is that we can carry out the procedure with the information *compressed* by an opportune data structure. But the limit of this approach is given by *how much* can we can compress data without losing any information.

A different approach was adopted in [4] by compressing information *too much*, i.e. by constructing an *approximation* of the model (an *abstraction*) and by using it to retrieve information on the solution of the original problem. This method is called CEGAR and it's an example of what in literature are called *abstraction-refinement methods*.

This is one of the fastest methods to solve some instances of the model checking problem, but the other face of the medal is that it solves only the *decisional* model checking problem for ACTL, namely the fragment of CTL defined by the following grammar:

$$\begin{aligned} \langle\varphi\rangle &\models \bot \mid p \in \mathtt{AP} \mid \neg p \mid \langle\varphi\rangle \wedge \langle\varphi\rangle \mid \langle\varphi\rangle \vee \langle\varphi\rangle \mid \mathtt{A}\langle\Phi\rangle \\ \langle\Phi\rangle &\models \mathtt{X}\langle\varphi\rangle \mid \mathtt{F}\langle\varphi\rangle \mid \mathtt{G}\langle\varphi\rangle \mid \langle\varphi\rangle \mathtt{U} \langle\varphi\rangle \end{aligned}$$

So these are CTL formulas using only the A quantifier with negation restricted to propositional formulas.

We present in this section the theory and main ideas behind CEGAR.

Definition 2.2.1 (CEGAR abstraction)**.** Let $\mathcal{M} = \langle S, \to, L\rangle$ be a KF and consider a partition $\mathcal{P}$ of S respecting the labeling function, i.e. such that

$$[s]_{\mathcal{P}} = [t]_{\mathcal{P}} \implies \forall p \in \mathtt{AP}.\, [s \in L(p) \iff t \in L(p)]$$

We define the *abstraction* of $\mathcal{M}$ relative to $\mathcal{P}$ as the KF $\mathcal{M}_{\mathcal{P}} = \langle \mathcal{P}, \to_{\mathcal{P}}, L_{\mathcal{P}}\rangle$ such that:

- $T \to_{\mathcal{P}} T' \iff$ There exist $t \in T$ and $t' \in T'$ such that $t \to t'$
- $T \in L_{\mathcal{P}}(p) \iff \exists t \in T. t \in L(p) \iff \forall t \in T. t \in L(p)$

Moral 2.2.2. The idea behind the abstraction presented above is that we can *identify* groups of states which satisfy the same propositional formulas. Note that in doing so we lose information on the global graph-structure of the original model.

Notation 2.2.3. From now on we will refer to $\mathcal{M}$ as the *concrete model* and to $\mathcal{M}_{\mathcal{P}}$ as the *abstract model* or *abstraction*.

There are several results that connect the two models. Here we introduce the bases of *Bisimulation and simulation theory* to better understand these connections.

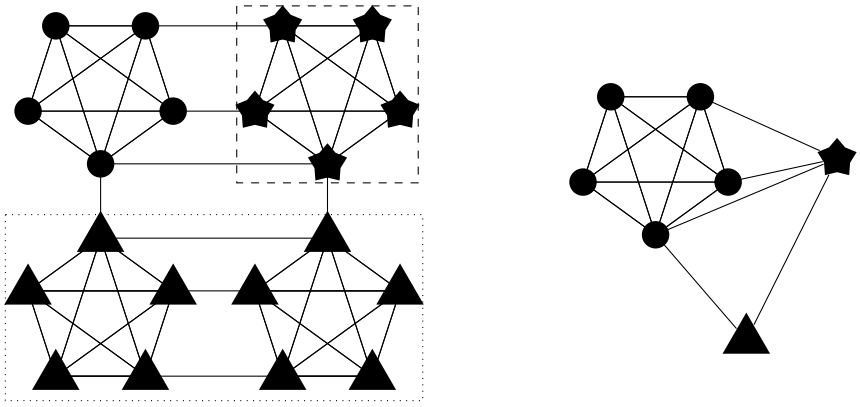


Figure 2.1: An example of abstraction for an undirected graph. The states with the same shape (on the left) are represented by a single node in the abstraction (on the right).

Definition 2.2.4 (Bisimulation and bisimilarity)**.** Given two `KFs` $\mathcal{M} = \langle S, \to, L\rangle$ and $\mathcal{M}' = \langle S', \to', L'\rangle$ we say that a relation $R \subseteq S \times S'$ is a bisimulation if:

1. $\forall s \in S. \exists s' \in S'. sRs'$
2. $\forall s' \in S'. \exists s \in S. sRs'$
3. $sRs' \implies \forall p \in \texttt{AP}. [s \in L(p) \iff s' \in L'(p)]$
4. $s \to t$ and $sRs' \implies$ there exists $t' \in S'$ such that $s' \to' t'$ and tRt'
5. $s' \to' t'$ and $sRs' \implies$ there exists $t \in S$ such that $s \to t$ and tRt'

If there exists a bisimulation R such that sRs', we say that s and s' are *bisimilar* and we indicate it with $s \sim s'$.

Moral 2.2.5. The idea behind the previous definition is that two states are bisimilar if *we can't distinguish them* in an operative way. The following technical lemma formalizes better this concept.

Lemma 2.2.6 (Paths and Bisimulations)**.** *Let $\mathcal{M}$ and $\mathcal{M}'$ be as in the definition above, let R be a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$. Then given a path of S*

$$\pi = (s_0, s_1, \dots)$$

there exists a path of S'

$$\pi' = (s'_0, s'_1, \dots)$$

such that $s_i R s'_i$ for all $i \in \mathbb{N}$.

Remark 2.2.7. Note that, by symmetry also the converse holds. Namely, given a path π' we can find a path π with the property above.

Lemma 2.2.8. *Consider the case $\mathcal{M}' = \mathcal{M}$. Then $\sim$ is an equivalence relation over S.*

Theorem 2.2.9. *Fix $\mathcal{M}$ and $\mathcal{M}'$ two* finite branching KFs *such that there exists at least a bisimulation $R \subseteq S \times S'$. Consider the relation $[\equiv_{CTL}] \subseteq S \times S'$ defined as*

$$s \equiv_{CTL} s' \iff \forall \varphi \in CTL. \left[\mathcal{M}, s \vDash \varphi \iff \mathcal{M}', s' \vDash \varphi\right]$$

then it holds $[\equiv_{CTL}] = [\sim]$.

Proof. To prove this result, we prove separately that $[\sim] \subseteq [\equiv_{\mathtt{CTL}}]$ and $[\equiv_{\mathtt{CTL}}] \subseteq [\sim]$.

- **We want to prove that for each formula φ it holds** $s \sim s' \implies [s \vDash \varphi \iff s' \vDash \varphi]$, and we can do so by induction over the structure of φ. Moreover, we can simplify the proof by using the *existential normal form* (so we don't have to deal with the operator A).

  - For $\varphi \equiv p \in \mathtt{AP}$ the result follows directly from the definition of bisimulation.
  - For φ obtained by applying a boolean operator, the result is trivial.
  - For $\varphi \equiv \mathtt{EQ}\psi$ for Q a path quantifier (or $\varphi \equiv \mathtt{E}(\psi_1 \mathtt{U} \psi_2)$) consider a path π of S such that $\mathcal{M}, \pi \vDash \mathtt{Q}\psi$ (respectively $\mathcal{M}, \pi \vDash \psi_1 \mathtt{U} \psi_2$). Then by Lemma 2.2.6 we have that there exists a path π' of S' such that

    $$\forall i \in \mathbb{N}. \left[\pi_i \sim \pi'_i\right] \implies \forall i \in \mathbb{N}. \left[\pi_i \vDash \psi \iff \pi'_i \vDash \psi\right]$$
    $$\forall i \in \mathbb{N}. \left[\pi_i \sim \pi'_i\right] \implies \forall i \in \mathbb{N}. \forall j \in \{1,2\}. \left[\pi_i \vDash \psi_j \iff \pi'_i \vDash \psi_j\right]$$

    Using this is trivial to prove that also $\mathcal{M}', \pi' \vDash \mathtt{Q}\psi$ (respectively $\mathcal{M}', \pi' \vDash \psi_1 \mathtt{U} \psi_2$).
    Using Remark 2.2.7 we have that also the converse holds, and so the result.

- **We want to prove that if $s \equiv_{\mathtt{CTL}} s'$ then it also holds $s \sim s'$.** To do so we claim that $\equiv_{\mathtt{CTL}}$ is a bisimulation. In fact:

  1. Consider $R \subseteq S \times S'$ a bisimulation (it exists by hypothesis). Then by the first part of this proof we have

     $$\left[\forall s \in S. \exists s' \in S'. sRs'\right] \implies \left[\forall s \in S. \exists s' \in S'. s \equiv_{\mathtt{CTL}} s'\right]$$

2. Same as the point above.
3. It follows from the fact that $s \equiv_{\mathtt{CTL}} s' \implies [s \vDash p \iff s' \vDash p]$.
4. Fix a state $s \in S$ and consider its successors $t_1, \ldots, t_k$ (finite by hypothesis). We then can consider the partition $\sqcup_{i=1}^{l} T_i = \{t_1, \ldots, t_k\}$ where T_i are the $\equiv_{\mathtt{CTL}}$ equivalence classes. We state here a technical lemma whose proof is omitted:

   Lemma 2.2.10. *Let $T_1, \ldots T_l$ as above. There exist formulas $\varphi_1, \ldots, \varphi_l$ such that*

   $$t_i \in T_j \iff t_i \vDash \varphi_j$$

   Moreover $\varphi_1, \ldots, \varphi_l$ depend only on the equivalence classes of $T_1, \ldots, T_l$.

   Given this, it's quite easy to show that

   $$s \vDash \mathtt{EX}(\varphi_i)$$
   $$s \vDash \mathtt{AX}\left(\bigvee_{i=1}^{l} \varphi_i\right)$$
   $$s \vDash \mathtt{AX}(\varphi_i \to \psi) \iff \forall t \in T_i.\, t \vDash \psi$$

   Now fix t_i and consider a generic $s' \in S'$ such that $s \equiv_{\mathtt{CTL}} s'$. We want to find a state t' such that $s' \to t'$ and $t_i \equiv_{\mathtt{CTL}} t'$.
   By the first and second properties above the successors of s' cover the same equivalence classes as the successors of s, and so we can fix t' a successor of s' such that $t' \vDash \varphi_i$. By the third property above, we have

   $$\begin{aligned} t_i \vDash \psi &\iff s \vDash \mathtt{AX}(\varphi_i \to \psi) \\ &\iff s' \vDash \mathtt{AX}(\varphi_i \to \psi) \\ &\iff t' \vDash \psi \end{aligned}$$

   where the last implication follows from the fact that $\varphi_1, \ldots, \varphi_l$ depends only on the equivalence classes of the T_i (again Lemma 2.2.10). Thus we have that $t_i \equiv_{\mathtt{CTL}} t'$ as wanted.
5. As the point above.

□

Corollary 2.2.11. *Given a finite KF $\mathcal{M}$, then $[\equiv_{CTL}] = [\sim]$.*

This last result give us a way to obtain a smaller KF preserving the same properties as the original one. Namely:

Definition 2.2.12 (Bisimulation Quotient)**.** Given a finite KF $\mathcal{M}$, define its *bisimulation quotient* as the KF $\mathcal{M}' = \langle S', \to', L' \rangle$ where:

- S' are the $[\sim]$-equivalence classes of S. In formulas:

$$S' = {}^{S}/_{\sim}$$

- $x' \to' y'$ if and only if for all $s \in x'$ there exists $t \in y'$ such that $x \to y$.
- $x' \in L'(p)$ if and only if $\forall s \in x'$, $s \in L(p)$.

Then, given $\pi : S \to S'$ the quotient map, for every CTL formula φ it holds:

$$s \vDash \varphi \iff \pi(s) \vDash \varphi$$

So what we obtained here is a "compacted model", but what we really wanted was an "approximation" of the original model. We present here another formal tool akin to bisimilarity, which we will use to prove the main properties of the CEGAR abstraction presented before.

Definition 2.2.13 (Simulation)**.** Given two KFs $\mathcal{M} = \langle S, \to, L \rangle$ and $\mathcal{M}' = \langle S', \to', L' \rangle$ we say that a relation $R \subseteq S \times S'$ is a *simulation* if:

1. $\forall s' \in S'. \exists s \in S. sRs'$
2. $sRs' \implies \forall p \in \mathtt{AP}. [s \in L(p) \iff s' \in L'(p)]$
3. $s \to t$ and $sRs' \implies$ there exists $t' \in S'$ such that $s' \to' t'$ and tRt'

If there exists a simulation R such that sRs', we say that s *simulates* s' and we indicate it with $s \succeq s'$.

Remark 2.2.14. Note that the properties above are the properties 1, 3 and 4 of Definition 2.2.4. Taking only these three breaks the symmetry of the previous definitions, and so $s \succeq s'$ can be intended as "s can perform every action that s' could do".

Now we list here the main theorems about simulation, corresponding to the ones presented before for bisimulation.

Lemma 2.2.15 (Paths and Simulations)**.** *Let $\mathcal{M}$ and $\mathcal{M}'$ be as in the definition above, let R be a simulation between $\mathcal{M}$ and $\mathcal{M}'$. Then given a path of S'*

$$\pi' = \left(s'_0, s'_1, \dots\right)$$

there exists a path of S

$$\pi = (s_0, s_1, \dots)$$

such that $s_i R s'_i$ for all $i \in \mathbb{N}$.

Remark 2.2.16. In this case, the converse doesn't hold.

Lemma 2.2.17. *Consider the case $\mathcal{M}' = \mathcal{M}$. Then $[\succeq] \cap [\preceq]$ is an equivalence relation over S.*

Theorem 2.2.18. *Fix $\mathcal{M}$ and $\mathcal{M}'$ two* finite branching `KFs` *such that there exists at least a simulation $R \subseteq S \times S'$. Then it holds:*

$$s \succeq s' \iff \text{for all } \varphi \in \texttt{ACTL} \text{ it holds } \left[s \vDash \varphi \implies s' \vDash \varphi\right]$$

Proof idea. This theorem corresponds to Theorem 2.2.9. The idea to prove it is exactly the same: consider a new relation

$$s \geq_{\texttt{ACTL}} s' \iff \forall \varphi \in \texttt{ACTL}. \left[s \vDash \varphi \implies s' \vDash \varphi\right]$$

and prove the two inclusions $[\succeq] \subseteq [\geq_{\texttt{ACTL}}]$ and $[\geq_{\texttt{ACTL}}] \subseteq [\succeq]$. □

Theorem 2.2.19. *Given a finite `KF` $\mathcal{M}$ and an abstraction $\mathcal{M}'$ as in Definition 2.2.1, consider the projection map $\pi : S \to S'$. Then it holds:*

- *π^{-1} is a simulation.*
- *For all $\varphi \in \texttt{ACTL}$, $\pi(s) \vDash \varphi$ implies $s \vDash \varphi$.*

Moral 2.2.20. The theorem above tells us *how* we can interpret an abstraction as an *approximation*. Note that it follows for $T \in \mathcal{M}'$ and $\varphi \in \texttt{ACTL}$

$$T \vDash \varphi \implies \forall t \in T. t \vDash \varphi$$

but is not necessarily true neither of the followings

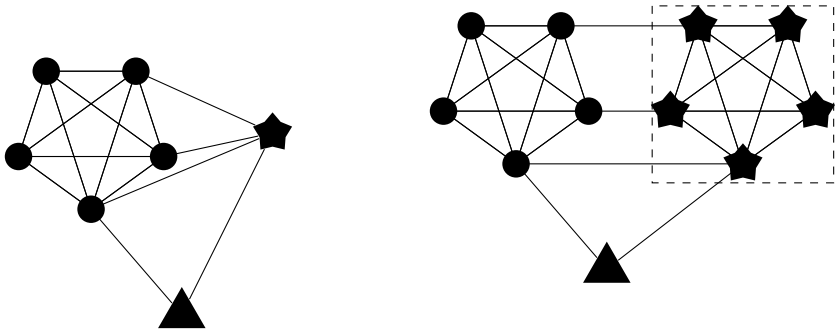
$$T \nvDash \varphi \implies \forall t \in T. t \nvDash \varphi$$
$$[\forall t \in T. t \vDash \varphi] \implies T \vDash \varphi$$

This last result can be used to create an efficient method to check if a single state of a model satisfies an `ACTL` formula.

Definition 2.2.21 (Expansion of abstract nodes)**.** Let $\mathcal{M}$ and $\mathcal{P}$ as in Definition 2.2.1. Fix $T_1, \ldots, T_k \in \mathcal{P}$. We define the partition obtained from $\mathcal{P}$ by *expanding* $T_1, \ldots, T_k$ as:

$$\mathcal{Q} = \{T \in \mathcal{P} | T \neq T_1, \ldots, T_k\} \cup \{\{t\} | t \in T_1 \cup \cdots \cup T_k\}$$

Example 2.2.22. Taking the abstraction of Figure 2.2, we can expand the "star" state obtaining:



Method 2.2.23. *Let $\mathcal{M} = \langle S, \to, L\rangle$ be a finite KF and fix a state $s \in S$ and an ACTL formula φ. The method consists of the following steps:*

1. *Choose a partition $\mathcal{P}$ and construct the associated abstraction $\mathcal{N} = \langle \mathcal{P}, \to', L'\rangle$. Let π be the quotient map.*
2. *Solve the MCP for $\mathcal{N}$, $\pi(s)$ and φ using the algorithm presented in Section 1.*
3. *If $\pi(s) \models \varphi$, then by Theorem 2.2.19 we have also $s \models \varphi$ and so we have finished.*
4. *Otherwise, if $\mathcal{P}$ is the trivial partition (i.e., $\mathcal{P} = \{\{s\}|s \in S\}$) then $s \not\models \varphi$ (as $\mathcal{M} = \mathcal{N}$, modulo identification of the states $s \in S$ and $\{s\} \in \mathcal{P}$).*
5. *If none of the previous apply,* choose *some non trivial abstract states $T_1, \ldots, T_k \in \mathcal{P}$ (i.e., $|T_i| > 1$) and expand them obtaining a new partition $\mathcal{Q}$. Repeat the procedure with $\mathcal{Q}$.*

Termination and correctness.

Termination If $\mathcal{P}$ is the trivial partition, then the algorithm terminates either giving back $s \models \varphi$ (step 3) or $s \not\models \varphi$ (step 4).

If $\mathcal{P}$ is not the trivial partition and the algorithm doesn't terminate at steps 3 and 4, then we have to choose a refinement $\mathcal{P}' \prec \mathcal{P}$ and repeat the algorithm (step 5). As $\mathcal{M}$ is finite, we can't have an infinite number of iterations as we would be constructing an infinite chain

$$\mathcal{P} \succ \mathcal{P}_1 \succ \mathcal{P}_2 \succ \ldots$$

Correctness The correctness follows from Theorem 2.2.19 (as indicated in step 3).

□

While this method terminates correctly, it has a major flaw: we can't estimate the time it takes to terminate.

There aren't theoretical results on the complexity of this algorithm, but by experimental results we can evince that:

- if it holds $s \models \varphi$ the time it takes is really small, it's actually faster then the linear algorithm in most of the cases. The reason is that if we choose well the abstract states $T_1, \ldots, T_k$, the algorithm terminates in just a few iterations, thus maintaining a small size for the models.
- if it holds $s \not\models \varphi$ then the algorithm takes a huge amount of time to terminate. The reason is simple: to decide $s \not\models \varphi$ it needs to obtain the trivial partition, thus it needs to solve several model checking problems relative to models of size comparable with the concrete one.

So this algorithm is slow! Moreover there are several missing points:

How do we choose the initial abstraction? The initial abstraction depends strongly on the problem at hand. For example, when we consider models obtained from parallel operators a "good" abstraction should preserve the parallel structure of the model.

How do we solve the speed problem? We need something more to speed-up the algorithm in case $s \not\models \varphi$.

How do we choose the states $T_1, \ldots, T_k$? As stated above, the algorithm is fast if we *choose well* the states $T_1, \ldots, T_k$. But how do we choose them? What does "well" means in this context?

As the first problem depends strongly on the initial model, we won't deal with it in this document, but we will tackle the other two problems. First of all, some new theory.

Definition 2.2.24 (ACTL counterexamples)**.**

- Let $\mathcal{M} = \langle S, \to, L \rangle$ be a KF and suppose that

  $$s \not\models \mathtt{A}\Psi$$

  for some state $s \in S$, for some ACTL formula $\mathtt{A}\Psi$. We define an *immediate counterexample* $\sigma_{\langle s,\Psi \rangle}$ for s and $\mathtt{A}\Psi$ as a path π starting at s such that

  $$\pi \not\models \Psi$$

- Let $\mathcal{M}, s$ as above, and consider a generic ACTL formula φ. We define a *counterexample tree* $\tau_{\langle s,\varphi \rangle}$ for s and φ by induction over the structure of φ as:

- If $\varphi \equiv \bot, p, \neg p$ then $\tau_{\langle s,\varphi\rangle}$ is the state s itself.
- If $\varphi \equiv \psi_1 \wedge \psi_2$ then $\tau_{\langle s,\varphi\rangle}$ is a pair $\langle i, \tau_{s,\psi_i}\rangle$ for i equal to 1 or 2.
- If $\varphi \equiv \psi_1 \vee \psi_2$ then $\tau_{\langle s,\varphi\rangle}$ is a pair $\left\langle \tau_{\langle s,\psi_1\rangle}, \tau_{\langle s,\psi_2\rangle}\right\rangle$.
- If $\varphi \equiv \mathtt{AX}(\psi)$ then $\tau_{\langle s,\varphi\rangle}$ is a pair $\langle\langle \pi, \tau_{\pi_1,\psi}\rangle\rangle$ where π is an immediate counterexample for s and $\mathtt{AX}(\psi)$.
- If $\varphi \equiv \mathtt{AF}(\psi)$ then $\tau_{\langle s,\varphi\rangle}$ is a pair $\left\langle \pi, (\tau_{\langle \pi_i,\psi\rangle})_{i\in\mathbb{N}}\right\rangle$ where π is an immediate counterexample for s and $\mathtt{AF}(\psi)$, and $(\tau_{\langle \pi_i,\psi\rangle})_{i\in\mathbb{N}}$ is a sequence of counterexample trees, one for each node of π.
- If $\varphi \equiv \mathtt{AG}(\psi)$ then $\tau_{\langle s,\varphi\rangle}$ is a pair $\left\langle \pi, \tau_{\langle \pi_i,\psi\rangle}\right\rangle$ for some $i \in \mathbb{N}$, where π is an immediate counterexample for s and $\mathtt{AG}(\psi)$.
- If $\varphi \equiv \mathtt{A}(\psi_1 \mathtt{U} \psi_2)$ then $\tau_{\langle s,\varphi\rangle}$ is a tuple $\left\langle \pi, (\tau_{\langle \pi_i,\psi_1\rangle})_{i<l}, \tau_{\langle \pi_l,\psi_2\rangle}\right\rangle$ where π is an immediate counterexample for s.

We will call the path π in the clauses AX, AF, AG and AU the *immediate counterexample associated to* $\tau_{\langle s,\varphi\rangle}$.

With a simple induction it's easy to prove the following:

Lemma 2.2.25. *Given $\mathcal{M}$, s and φ as above, the followings are equivalent:*

- $s \nVdash \varphi$
- *There exists an immediate counterexample for s and φ.*
- *There exists a counterexample tree for s and φ.*

Moral 2.2.26. The idea behind this result is really simple. An ACTL formula φ tells us that *all paths* starting at a point have some property. An immediate counterexample is a path that *doesn't have* that property. Instead a counterexample tree gives us more information, namely *why* that path doesn't entail the formula, and it does so by giving us counterexample trees for some states of the path.

We don't present in this document an algorithm to find a counterexample tree, but a simple modification of the algorithms presented in 1 can achieve this result while performing the model checking procedure.

Now, consider the case introduced before where we are dealing with a model $\mathcal{M}$ and one of its abstractions $\mathcal{N}$, and we are trying to solve the MCP for s and φ. If we indicate with $[s]$ the class of s in $\mathcal{N}$, and we have that $[s] \nVdash \varphi$, then we can find an immediate counterexample for $[s]$ and φ. But does this correspond to an immediate counterexample for s? Formally:

Definition 2.2.27. Given a path π in $\mathcal{N}$ we call a *lifting of* π to $\mathcal{M}$ a path π' of $\mathcal{M}$ such that

$$[\pi'_i] = \pi_i$$

If there exists such a path π' then we say that π *lifts* to $\mathcal{M}$. Moreover, if $\pi'_0 = s$ we say that π lifts with *base point* s.

So we are asking, does π lifts to $\mathcal{M}$ with base point s? There are three cases:

- If π' is a lifting of π *and* is a counterexample for s and φ, then it follows from Lemma 2.2.25 that $s \not\models \varphi$.
- If π doesn't lift to $\mathcal{M}$, then we can't decide if $s \models \varphi$ or not (there could be other counterexamples!), but we know *why* π doesn't lift. Namely, there is a set of states $\mathcal{T} \subseteq S_{\mathcal{N}}$ such that
  - The finite path $\langle \pi_0, \ldots, \pi_l \rangle$ lifts to $\mathcal{M}$, but $\langle \pi_0, \ldots, \pi_{l+1} \rangle$ doesn't.
  - The last states of the possible lifted paths of $\langle \pi_0, \ldots, \pi_l \rangle$ are in $\bigcup \mathcal{T}$.
  - $\mathcal{T}$ is the minimal set with these properties.

  In this case we will call $\mathcal{T}$ the *critical set* and say that π is a *spurious counterexample of the first type.*
- If $s \models \varphi$ then there are no immediate counterexamples for s and φ, but *there could be a lifting* π' *of* π. Of course, in this case π' *is not* a counterexample and so we will say that π is a *spurious counterexample of the second type.*

Now we are ready to present the `CEGAR` method as presented in [4].

Method 2.2.28. *Let $\mathcal{M} = \langle S, \to, L \rangle$ be a finite `KF`. Fix a state $s \in S$ and an `ACTL` formula φ. The method consists of the following steps:*

1. *Choose a partition $\mathcal{P}$ and construct the associated abstraction $\mathcal{N} = \langle \mathcal{P}, \to', L' \rangle$. Let π be the quotient map.*
2. *Solve the `MCP` for $\mathcal{N}$, $\pi(s)$ and φ using the algorithm presented in Section 1. Moreover, if $s \not\models \varphi$ find an immediate counterexample σ.*
3. *If $\pi(s) \models \varphi$, then by Theorem 2.2.19 we have also $s \models \varphi$ and so we are done.*
4. *Otherwise check if σ lifts to $\mathcal{M}$. If it doesn't lift (i.e., σ is a counterexample of the first type), then find the critical set $\mathcal{T}$, expand the nodes in $\mathcal{T}$ and repeat the procedure.*
5. *If it does lift to a path σ', check if σ' is a counterexample.*
   (a) *If σ' is a counterexample then we finished as $s \not\models \sigma$ by Theorem 2.2.19.*
   (b) *If σ' is not a counterexample...*

This complete the method *except* for the step 5*b*. In this case we need a more convoluted definition of critical set, in particular a definition for counterexamples of the second type.

Definition 2.2.29 (Critical set)**.** Given $\tau = \tau_{\langle s,\varphi\rangle}$ a counterexample tree for the state s and the formula φ, we define its *critical set* $\mathcal{T}(\tau)$ by induction over the structure of φ:

- If $\varphi \equiv \bot, p, \neg p$ then $\mathcal{T}(\tau) = \{s\}$.
- If $\varphi \equiv \psi_1 \wedge \psi_2$ then $\mathcal{T}(\tau) = \mathcal{T}(\tau_{\langle s,\psi_i\rangle})$ where $\tau = \langle i, \tau_{\langle s,\psi_i\rangle}\rangle$.
- If $\varphi \equiv \psi_1 \vee \psi_2$ then $\mathcal{T}(\tau_{\langle s,\psi_2\rangle}) \cup \mathcal{T}(\tau_{\langle s,\psi_2\rangle})$.
- If $\varphi \equiv \mathtt{AX}(\psi)$, let $\tau = \langle \pi, \tau_{\langle \pi_1,\psi\rangle}\rangle$. There are two cases:
  - if π doesn't lift, then $\mathcal{T}(\tau) = \{\pi_0\}$
  - else $\mathcal{T}(\tau) = \mathcal{T}(\tau_{\langle \pi_1,\psi\rangle})$.
- If $\varphi \equiv \mathtt{AF}(\psi)$, let $\tau = \langle \pi, (\tau_{\langle \pi_i,\psi\rangle})_{i\in\mathbb{N}}\rangle$. There are two cases:
  - if π doesn't lift, then $\mathcal{T}(\tau) = \{\pi_k\}$ where π_k is the only state such that $\langle \pi_0, \ldots, \pi_k\rangle$ lifts but $\langle \pi_0, \ldots, \pi_{k+1}\rangle$ doesn't lift.
  - else $\mathcal{T}(\tau) = \bigcup_{i\in\mathbb{N}} \mathcal{T}(\tau_{\langle \pi_i,\psi\rangle})$
- If $\varphi \equiv \mathtt{AG}(\psi)$, let $\tau = \langle \pi, \tau_{\langle \pi_h,\psi\rangle}\rangle$. Then
  - if $\langle \pi_0, \ldots, \pi_h\rangle$ doesn't lift, then $\mathcal{T}(\tau) = \{\pi_k\}$ where π_k is the only state such that $\langle \pi_0, \ldots, \pi_k\rangle$ lifts but $\langle \pi_0, \ldots, \pi_{k+1}\rangle$ doesn't lift.
  - else $\mathcal{T}(\tau) = \mathcal{T}(\tau_{\langle \pi_h,\psi\rangle})$
- If $\varphi \equiv \mathtt{A}(\psi_1 \mathtt{U} \psi_2)$, let $\tau = \langle \pi, (\tau_{\langle \pi_i,\psi_1\rangle})_{i<l}, \tau_{\langle \pi_l,\psi_2\rangle}\rangle$. Then:
  - if $\langle \pi_0, \ldots, \pi_l\rangle$ doesn't lift, then $\mathcal{T}(\tau) = \{\pi_k\}$ where π_k is the only state such that $\langle \pi_0, \ldots, \pi_k\rangle$ lifts but $\langle \pi_0, \ldots, \pi_{k+1}\rangle$ doesn't lift.
  - else $\mathcal{T}(\tau) = \bigcup_{i\leq(l-1)} \mathcal{T}(\tau_{\langle \pi_i,\psi_1\rangle}) \cup \mathcal{T}(\tau_{\langle \pi_l,\psi_2\rangle})$.

Moral 2.2.30. As the critical set introduced for an immediate counterexample π tells us why π doesn't lift (and so it's a spurious counterexample), the critical set introduced here tells us why *the entire tree* doesn't lift. So for each branch we have a critical point, and the critical set of the tree collects those critical points.

We present now the revisited version of CEGAR:

Method 2.2.31. *Let $\mathcal{M} = \langle S, \to, L\rangle$ be a finite KF. Fix a state $s \in S$ and an ACTL formula φ. The method consists of the following steps:*

1. *Choose a partition $\mathcal{P}$ and and construct the associated abstraction $\mathcal{N} = \langle \mathcal{P}, \rightarrow', L' \rangle$. Let π be the quotient map.*

2. *Solve the* `MCP` *for $\mathcal{N}$, $\pi(s)$ and φ using the algorithm presented in Section 1. Moreover, if $s \not\models \varphi$ find a counterexample tree τ and let σ be its associated immediate counterexample.*

3. *If $\pi(s) \models \varphi$, then by Theorem 2.2.19 we have also $s \models \varphi$ and so we finished.*

4. *Otherwise check if σ lifts to $\mathcal{M}$. If it doesn't lift (i.e., σ is a counterexample of the first type), then find the critical set $\mathcal{T}$ of σ, expand the nodes in $\mathcal{T}$ and repeat the procedure.*

5. *If σ does lift, check if τ lifts.*

   (a) *If τ lifts then we finished as $s \not\models \sigma$ by Theorem 2.2.19.*

   (b) *If τ doesn't lift (i.e., σ is a counterexample of the second type), then find the critical set $\mathcal{T}$ of τ, expand the nodes in $\mathcal{T}$ and repeat the procedure.*

Moral 2.2.32. The idea behind this method is that, if we find a counterexample tree τ in the abstract model, we want to check if it corresponds to an actual counterexample in the concrete model. If it does, then we have a counterexample and so we finished. Otherwise, by expanding the critical set we *eliminate* τ from the abstract model.

2.3 Abstraction-Refinement for μ-Calculus

We quickly describe here a generalization of the `CEGAR` approach presented in [12] to the full μ-calculus. To do so we need to introduce a new mathematical object.

Definition 2.3.1 (Kripke Modal Transition System)**.** A *Kripke modal transition system* (or `KMTS`) is a tuple $\mathcal{M} = \langle S, \dashrightarrow, \rightarrow, L \rangle$ where:

- S is a set of states.
- $\dashrightarrow, \rightarrow \subseteq S \times S$ are accessibility relations (called *may transitions* and *must transitions* respectively) such that $[\rightarrow] \subseteq [\dashrightarrow]$.
- $L : S \times \mathtt{AP} \rightarrow \{\bot, ?, \top\}$ is a *3-valued evaluation function* (meaning we will consider $\bot$, ? and $\top$ as truth values).

In particular, we can identify `KFs` as `KMTSs` such that $\rightarrow = \dashrightarrow$ and L assumes only values $\top$ and $\bot$.

Moral 2.3.2. A KMTS represents a KF that is not completely determined. In particular, we have some information that is certain (a must transition represent a "transition that must be there"; if $L(p,s) = \top$ then "p has to be true at s") and some that is not certain (a may transition represent a "transition that may or may not be there"; if $L(p,s) = ?$ we "don't know the value of p at s").

With KMTSs we can define a more convoluted abstraction, namely:

Definition 2.3.3 (Modal Abstraction)**.** Let $\mathcal{M} = \langle S, \rightarrow, L\rangle$ be a KF and $\mathcal{P}$ be a partition of S. Then we define the *modal abstraction* of $\mathcal{M}$ with respects of $\mathcal{P}$ (or simply abstraction) as the KMTS $\mathcal{N} = \langle \mathcal{P}, \dashrightarrow, \rightarrow, L'\rangle$ such that:

- $\dashrightarrow$ and $\rightarrow$ are defined as

$$T \dashrightarrow T' \iff \text{there exists } t \in T \text{ and } t' \in T' \text{ such that } t \rightarrow t'$$
$$T \rightarrow T' \iff \text{for all } t \in T \text{ there exists } t' \in T' \text{ such that } t \rightarrow t'$$

  Note that $\rightarrow \subseteq \dashrightarrow$.

- L' is defined as

$$L'(p,T) = \top \iff \text{for all } t \in T \text{ it holds } L(p,t) = \top$$
$$L'(p,T) = \bot \iff \text{for all } t \in T \text{ it holds } L(p,t) = \bot$$

  otherwise $L'(p,t) = ?$.

Now we proceed as in the previous section, by showing that we have a *transfer property* that binds the semantics of a formula evaluated over the abstraction and in the concrete model.

Definition 2.3.4 (3-Valued Environment)**.** Given $\mathcal{M} = \langle S, \dashrightarrow, \rightarrow, L\rangle$ a KMTS, we call a function

$$g : \mathtt{Var} \times S \rightarrow \{\top, ?, \bot\}$$

a *3-valued environment* for $\mathcal{M}$. With $g[X \mapsto f]$ we indicate the environment such that

$$g[X \mapsto f](Y,s) = \begin{cases} g(Y,s) & \text{if } Y \neq X \\ f(s) & \text{if } Y = X \end{cases}$$

Definition 2.3.5 (KMTS semantics for μ-calculus)**.** Let $\mathcal{M} = \langle S, \dashrightarrow, \rightarrow, L\rangle$ be a KMTS, $g : \mathtt{Var} \times S \rightarrow \{\top, ?, \bot\}$ a 3-valued environment and φ, a μ-calculus formula. We want to define what it means that φ has value θ (with $\theta \in \{\bot, ?, \top\}$) at a state s under the environment g, in symbols $\mathcal{M}, s \models^g_\theta \varphi$.

To simplify the next definition we introduce the notation $[\![\varphi]\!]^g_\theta$ for the set of states s such that $\mathcal{M}, s \models^g_\theta \varphi$ and $Sem(\varphi)^g : S \rightarrow \{\bot, ?, \top\}$ for the function that associates to each $s \in S$ the only θ such that $s \models^g_\theta \varphi$.

By induction over the formula φ we define:

$$
\begin{aligned}
[\![\bot]\!]^g_\theta &= \begin{cases} \emptyset & \text{if } \theta = \top, ? \\ S & \text{if } \theta = \bot \end{cases} \\
[\![p]\!]^g_\theta &= \{s \in S | L(p,s) = \theta\} \\
[\![X]\!]^g_\theta &= \{s \in S | g(X,s) = \theta\} \\
[\![\neg\psi]\!]^g_\theta &= [\![\psi]\!]^g_{\neg\theta} \\
[\![\psi_1 \wedge \psi_2]\!]^g_\theta &= \bigcup_{\theta_1 \wedge \theta_2 = \theta} [\![\psi_1]\!]^g_{\theta_1} \cap [\![\psi_2]\!]^g_{\theta_2} \\
[\![\Box\,\psi]\!]^g_\top &= \left\{ s \in S | \forall t.\, \left[s \dashrightarrow t \implies t \vDash^g_\top \psi \right] \right\} \\
[\![\Box\,\psi]\!]^g_\bot &= \left\{ s \in S | \exists t.\, \left[s \to t \wedge t \vDash^g_\bot \psi \right] \right\} \\
[\![\Box\,\psi]\!]^g_? &= S \setminus \left([\![\Box\,\psi]\!]^g_\top \cup [\![\Box\,\psi]\!]^g_\bot \right) \\
[\![\Diamond\psi]\!]^g_\top &= \left\{ s \in S | \exists t.\, \left[s \dashrightarrow t \wedge t \vDash^g_\top \psi \right] \right\} \\
[\![\Diamond\psi]\!]^g_\bot &= \left\{ s \in S | \exists t.\, \left[s \to t \implies t \vDash^g_\bot \psi \right] \right\} \\
[\![\Diamond\psi]\!]^g_? &= S \setminus \left([\![\Diamond\psi]\!]^g_\top \cup [\![\Diamond\psi]\!]^g_\bot \right) \\
Sem\,(\mu X.\psi)^g &= \bigwedge \left\{ f \,\middle|\, Sem\,(\varphi)^{g[X \mapsto f]} \leq f \right\} \\
Sem\,(\nu X.\psi)^g &= \bigvee \left\{ f \,\middle|\, Sem\,(\varphi)^{g[X \mapsto f]} \geq f \right\}
\end{aligned}
$$

Moral 2.3.6. The definition above is quite complicated, but the idea behind it is simple: a state s thinks that a formula φ is true if and only if is true for every possible instantiation of the model. Meaning that if we *choose* which may transitions are real transitions and which are not, and if we *choose* which nodes entail which propositions, then we obtain a `KF` such that $s \vDash_g \varphi$ independently from the choices made.

Lemma 2.3.7. *Using the notation of the definition above:*

- *The semantics presented is well-defined and given a state s there exists one and only one $\theta \in \{\bot, ?, \top\}$ such that $s \vDash^g_\theta \varphi$.*
- *The semantics above extends the semantics already introduced for μ-calculus. Meaning that given a `KF`, the two semantics coincide.*

With techniques similar to the one used for `CEGAR` we have:

Theorem 2.3.8. *Given $\mathcal{M}$ a `KF` and $\mathcal{N}$ the abstraction obtained from a partition $\mathcal{P}$, then it holds for φ a closed μ-calculus formula:*

$$
\begin{aligned}
\mathcal{N}, T \vDash_\top \varphi &\implies \forall t \in T.\, \mathcal{M}, t \vDash \varphi \\
\mathcal{N}, T \vDash_\bot \varphi &\implies \forall t \in T.\, \mathcal{M}, t \nvDash \varphi
\end{aligned}
$$

Moreover we can define a *simulation relation* between `KMTSs` similar to the `KF` one.

Definition 2.3.9 (Modal simulation)**.** Let $\mathcal{M} = \langle S, \dashrightarrow, \to, L\rangle$ and mbox$\mathcal{M}' = \langle S', \dashrightarrow', \to', L'\rangle$ be two KMTSs. We say that a relation $R \subseteq S \times S'$ is a *simulation* between $\mathcal{M}$ and $\mathcal{M}'$ if the followings hold:

- For each $s' \in S'$ there exists $s \in S$ such that sRs'.
- If $L(p, s) = \top$ and sRs' then $L'(p, s') = \top$.
- If $L(p, s) = \bot$ and sRs' then $L'(p, s') = \bot$.
- If $s \to t$, sRs' and tRt' then $s' \to' t'$.
- If $s' \dashrightarrow' t'$, sRs' and tRt' then $s \dashrightarrow t$.

We say that $\mathcal{M}$ *simulates* $\mathcal{M}'$ (and we indicate it with $\mathcal{M} \succeq \mathcal{M}'$) if there exists a simulation between $\mathcal{M}$ and $\mathcal{M}'$. Given two states $s \in S$ and $s' \in S'$, we say that s *simulates* s' (and we indicate it with $s \succeq s'$) if there exists a simulation between $\mathcal{M}$ and $\mathcal{M}'$ such that sRs'.

Moral 2.3.10. The idea is that "potentially" $\mathcal{M}$ could do everything that $\mathcal{M}'$ can, It depends on how the may transitions and the propositions are instantiated.

Theorem 2.3.11. *Let $\mathcal{M}$ and $\mathcal{M}'$ as in the definition above. Let $s \in S$ and $s' \in S'$ such that $s \succeq s'$. Then it holds for φ closed:*

$$\mathcal{M}, s \vDash_\top \varphi \implies \mathcal{M}', s' \vDash_\top \varphi$$
$$\mathcal{M}, s \vDash_\bot \varphi \implies \mathcal{M}', s' \vDash_\bot \varphi$$

Corollary 2.3.12. *Let $\mathcal{M}$ be a KF and consider two partitions of its states $\mathcal{P} \preceq \mathcal{Q}$. Then, given the abstractions $\mathcal{M}_\mathcal{P}$ and $\mathcal{M}_\mathcal{Q}$ obtained from $\mathcal{P}$ and $\mathcal{Q}$ respectively, it holds:*

$$\mathcal{M}_\mathcal{P} \preceq \mathcal{M}_\mathcal{Q}$$

In particular we have

$$\mathcal{M} \preceq \mathcal{M}_\mathcal{Q}$$

This results gives us a way to adapt the CEGAR approach to the full μ-calculus. Namely:

Method 2.3.13. *Let $\mathcal{M} = \langle S, \to, L\rangle$ be a finite KF and fix a state $s \in S$ and a closed μ-calculus formula φ. The method consists of the following steps:*

1. *Choose a partition $\mathcal{P}$ and construct the associated abstraction $\mathcal{N} = \langle \mathcal{P}, \dashrightarrow, \to, L'\rangle$. Let π be the quotient map.*
2. *Solve the MCP for $\mathcal{N}$, $\pi(s)$ and φ.*
3. *If $\pi(s) \vDash_\top \varphi$ or $\pi(s) \vDash_\bot \varphi$, then by Theorem 2.3.11 and Corollary 2.3.12 we have $s \vDash \varphi$ or $s \nvDash \varphi$ respectively and so we finished.*

4. *Otherwise* choose *some non trivial abstract states* $T_1, \ldots, T_k \in \mathcal{P}$ *and expand them obtaining a new partition* $\mathcal{Q}$. *Repeat the procedure with* $\mathcal{Q}$.

Several question arise from this method:

- How do we choose the initial abstraction?
- How do we solve the model checking problem for a `KMTS`?
- How do we choose the states to expand?
- Is this method efficient?

For a thorough answer to these questions, we refer to [12]. What we want to point here are the advantages of expanding the semantics to consider uncertainty (i.e., by considering the truth value ? and the may transitions). What we gain is that we can work with approximations and the full μ-calculus logic without changing the overall structure of the method.

In the following chapters, we will try to apply a similar approach to a more specific problem, the `SLCS` model checking problem, by abstracting the state space and considering a 3-valued semantics as the one presented above.

Chapter 3

Spatio-Temporal Logics and Model Checking

An interesting research field currently under development is that of *spatio-temporal logics*.

A temporal logic is any system of rules for representing, and reasoning about, propositions qualified in terms of time. Examples of these logics were introduced in the previous chapters: `CTL` and μ-calculus can be interpreted as temporal logics if we consider the models as discrete descriptions of some event. And in several cases is quite trivial to do so, for example when we consider the system representing a program, the accessibility relation connect two states whenever one can be obtained from the other *after* the execution of the program.

In a similar way, a spatial logic is "any formal language interpreted over a class of structures featuring geometrical entities and relations, broadly construed" ([13]). There are several examples of spatial logics (Euclidean geometry, mereotopologies, spatial modal logics, ...) and several applications of these (for example, in the field of artificial intelligence *qualitative spatial reasoning logics* such as $\mathcal{RCC}$-8 are broadly studied [14]).

Spatio-temporal logics are logics that represent both the temporal and spatial structure of a system, and more importantly the *connections* between those two heterogeneous structures. The main problem in this case is finding the right balance between *expressivity* and *decidability*, as there are several examples of these logics which are *not* recursively axiomatizable (as an example take *Dynamic Topological Logic* [13]).

In this section we will present and study the spatial logic `SLCS` (presented in [5] and [15]) and its generalization to a spatio-temporal logic `STLCS` (presented in [15] and [6]). Both logics represent the spatial structure of a system as a *closure space* (a generalization of topological spaces), introducing thus an asymmetric concept of "proximity". The logic `STLCS` then combines this representation with the temporal logic `CTL` to describe convoluted spatial

and temporal properties of the system, such as *eventual reachability* and *definitive safety properties*.

In particular we will focus on solving the model checking problem for these two logics. An actual implementation of all the algorithms presented in this chapter can be found in [16].

3.1 Spatial Logic of Closure Spaces

Definition 3.1.1 (Syntax of SLCS)**.** The syntax of SLCS formulas is defined by the following grammar:

$$\langle\varphi\rangle \quad ::= \quad \bot \mid p \in \mathtt{AP} \mid \langle\varphi\rangle \wedge \langle\varphi\rangle \mid \neg\langle\varphi\rangle \mid \mathcal{S}(\langle\varphi\rangle, \langle\varphi\rangle) \mid \mathcal{N}(\langle\varphi\rangle)$$

As models for this logic we will consider *closure spaces*, mathematical objects that generalize topological spaces.

Definition 3.1.2 (Closure Space)**.** A *closure space* is a pair $\langle X, \mathcal{C}\rangle$ where X is a set and $\mathcal{C} : \mathcal{P}(X) \to \mathcal{P}(X)$ has the following properties:

$$\begin{aligned} \textbf{(}\boldsymbol{\emptyset}\textbf{)} \quad & \mathcal{C}(\emptyset) = \emptyset \\ \textbf{(union)} \quad & \mathcal{C}(A \cup B) = \mathcal{C}(A) \cup \mathcal{C}(B) \\ \textbf{(expansion)} \quad & A \subseteq \mathcal{C}(A) \end{aligned}$$

We will call $\mathcal{C}$ a *closure operator* for X.

Remark 3.1.3. Note the resemblance with the Kuratowski closure operator for a topology. In fact, taking a closure space with the following property

$$\textbf{(idempotence)} \quad \mathcal{C}(\mathcal{C}(A)) = \mathcal{C}(A)$$

we obtain a *Kuratowski closure operator*, and so by taking the complementary of the subsets in $\mathcal{C}[\mathcal{P}(X)]$ as the open sets we define a topology.

Of course not all closure spaces are topological spaces, as shown in the example below.

Example 3.1.4. Take $X = \mathbb{R}^2$ and consider as the closure operator:

$$\mathcal{C}(A) = \{x \in X | d(A, x) \leq 1\}$$

where d is the usual euclidean distance. It's easy to check that this is a closure space, but the operator $\mathcal{C}$ is not idempotent (see figure 3.1.4).

The remark above suggests the following definitions:

Definition 3.1.5. Consider $\langle X, \mathcal{C}\rangle$ two closure spaces.

- We define a *closed set* as an element of $\mathcal{C}[\mathcal{P}(X)]$

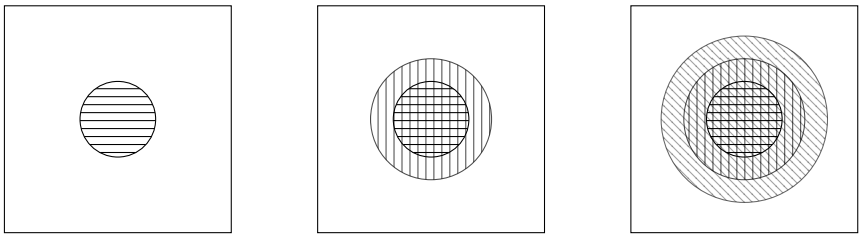


Figure 3.1: $\langle \mathbb{R}^2, \mathcal{C} \rangle$ as a closure space. In the figure we have a subset A (horizontal lines) and its iterated closures $\mathcal{C}(A)$ and $\mathcal{C}(\mathcal{C}(A))$ (vertical and oblique lines respectively).

- We define the *interior operator* as

$$\mathcal{I}(A) = \overline{\mathcal{C}(\overline{A})}$$

- We define an *open set* as an element of $\mathcal{I}[\mathcal{P}(X)]$

Closure spaces retain some of the properties of topological spaces. Here we give some examples:

Lemma 3.1.6. *For $\langle X, \mathcal{C} \rangle$ a closure space:*

- *The interior operator satisfies:*

$$\begin{array}{rl} \textbf{(}\boldsymbol{X}\textbf{)} & \mathcal{I}(X) = X \\ \textbf{\textit{(intersection)}} & \mathcal{I}(A \cap B) = \mathcal{I}(A) \cap \mathcal{I}(B) \\ \textbf{\textit{(contraction)}} & \mathcal{I}(A) \subseteq A \end{array}$$

- ***monotonicity:*** *If $A \subseteq B$ then $\mathcal{C}(A) \subseteq \mathcal{C}(B)$ and $\mathcal{I}(A) \subseteq \mathcal{I}(B)$.*
- *A set A is closed if and only if its complement $\overline{A}$ is open.*
- *Finite union and arbitrary intersection of closed sets is closed. Finite intersection and arbitrary union of open sets is open.*

The following definitions will prove to be useful to describe the semantics of `SLCS`

Definition 3.1.7. Given $\langle X, \mathcal{C} \rangle$ a closure space and $A \subseteq X$ we define:

$$\mathcal{B}^{+}(A) = \mathcal{C}(A) \setminus A$$
$$\mathcal{B}^{-}(A) = A \setminus \mathcal{I}(A)$$

Definition 3.1.8 (`SLCS` model)**.**

- Given $\langle X, \mathcal{C} \rangle$ a closure space, a *valuation* over X is a function $g : \mathtt{AP} \to \mathcal{P}(X)$.

- We define a *closure space with valuation* (or simply a *model*) for `SLCS` to be a tuple $\mathcal{X} = \langle X, \mathcal{C}, g \rangle$ where $\langle X, \mathcal{C} \rangle$ is a closure space and g is a valuation over X.

Now we proceed to give the semantics for the logic.

Definition 3.1.9 (Closure space semantics for `SLCS`)**.** Given a model $\mathcal{X} = \langle X, \mathcal{C}, g \rangle$, we want to define what it means that a state $s \in X$ *entails* a formula φ (in symbols $\mathcal{X}, s \vDash \varphi$). To simplify the next definition we introduce the notation $[\![\varphi]\!]^{\mathcal{X}}$ for the set of states that entails φ. By induction over the formula φ we define:

$$
\begin{aligned}
&\mathcal{X}, s \nvDash \bot \\
&\mathcal{X}, s \vDash p \in \mathtt{AP} && \iff s \in g(p) \\
&\mathcal{X}, s \vDash \varphi_1 \wedge \varphi_2 && \iff \mathcal{X}, s \vDash \varphi_1 \text{ and } \mathcal{X}, s \vDash \varphi_2 \\
&\mathcal{X}, s \vDash \neg\varphi && \iff \mathcal{X}, s \nvDash \varphi \\
&\mathcal{X}, s \vDash \mathcal{S}(\varphi_1, \varphi_2) && \iff \text{there exists } A \subseteq X \text{ such that } s \in A \\
& && \qquad \text{and } \begin{cases} A & \subseteq & [\![\varphi_1]\!]^{\mathcal{X}} \\ \mathcal{B}^+(A) & \subseteq & [\![\varphi_2]\!]^{\mathcal{X}} \end{cases} \\
&\mathcal{X}, s \vDash \mathcal{N}(\varphi) && \iff s \in \mathcal{C}\left([\![\varphi]\!]^{\mathcal{X}}\right)
\end{aligned}
$$

Moral 3.1.10. The new ingredients that this logic introduces are the operators $\mathcal{S}$ and $\mathcal{N}$. How do we interpret them?

- $\mathcal{S}$**:** This operator describes a *safety condition*. $\mathcal{S}(p, q)$ states that the state is in a zone where p holds and which is boarded by a zone where q holds. With an appropriate definition of "path", it can be interpreted as "we can't reach a point where p doesn't hold until we reach a point where q holds".
- $\mathcal{N}$**:** This operator represents the closure operator of the space.

Example 3.1.11. Take the closure space $\left\langle \mathbb{R}^2, \mathcal{C} \right\rangle$ as in the Example 3.1.4 and consider the following examples of semantics.

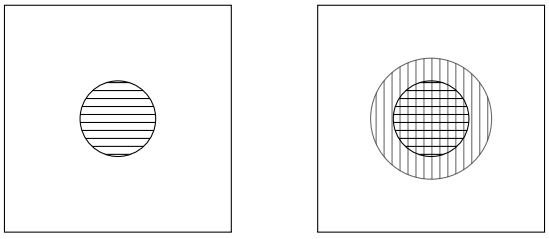


Figure 3.2: On the left the semantics of p (horizontal lines). On the right the semantics of $\mathcal{N}(p)$ (vertical lines).

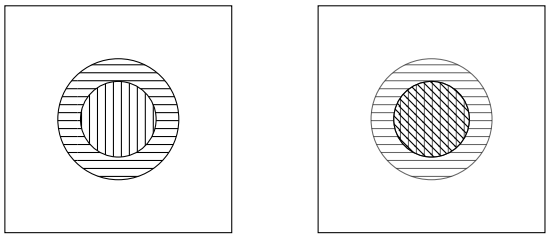


Figure 3.3: On the left the semantics of p and q (vertical and horizontal lines respectively). On the right the semantics of $\mathcal{S}(p, q)$ (oblique lines).

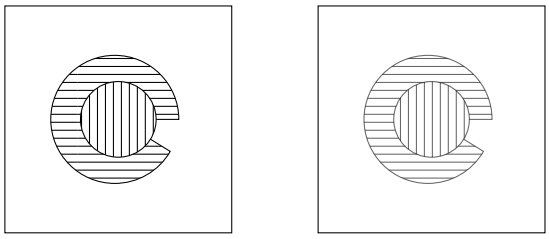


Figure 3.4: On the left the semantics of p and q (vertical and horizontal lines respectively). On the right the semantics of $\mathcal{S}(p, q)$ (nothing changed as $[\![\mathcal{S}(p, q)]\!] = \emptyset$). This happens because there is no p zone completely surrounded by a q zone.

Remark 3.1.12. The interpretation of the $\mathcal{S}$ operator is quite tricky. In general we have that each state in $[\![\mathcal{S}(p, q)]\!]$ is in a "safe zone" (meaning a zone where p holds and whose border entails q), but such zone *may not be* $[\![\mathcal{S}(p, q)]\!]$ itself, as the following example shows.

Example 3.1.13. Consider the tuple $\langle \omega + 1, \mathcal{C}, g \rangle$ where:

- $\omega + 1 = \omega \cup \{\omega\} = \{0, \ldots, n, \ldots; \omega\}$ is the second transfinite Von Neumann ordinal.
- $\mathcal{C}(A) = A \cup \{n + 1 \in \omega | n \in A\} \cup \begin{cases} \{\omega\} & \text{if } A \text{ is unbounded in } \mathbb{N} \\ \emptyset & \text{otherwise} \end{cases}$
- $g(p) = \omega$

It's easy to verify this is a closure space:

($\emptyset$): Using the definition is immediate to show $\mathcal{C}(\emptyset) = \emptyset$.

(expansion): by definition of $\mathcal{C}$.

(union): notice that $A \cup B$ is unbounded in $\mathbb{N}$ if and only if either A or B is unbounded in $\mathbb{N}$, and that

$$\{n + 1 | n \in A \cup B\} = \{n + 1 | n \in A\} \cup \{n + 1 | n \in B\}$$

So it follows that $\mathcal{C}(A \cup B) = \mathcal{C}(A) \cup \mathcal{C}(B)$.

In particular it holds:

- $[\![\mathcal{S}(p,p)]\!] = \omega$ as for every $n \in \omega$ we have $\mathcal{C}(\{n\}) = \{n, n+1\} \subseteq [\![p]\!]$, while $\omega \notin [\![p]\!]$.
- $\{\omega\} = \mathcal{B}^+([\![\mathcal{S}(p,p)]\!])$, and so $[\![\mathcal{S}(p,p)]\!]$ is not a "safe zone".

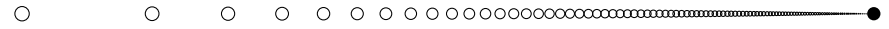


Figure 3.5: A visual interpretation of $\omega + 1$ (the last point being ω). In white the points for which p holds.

Do we know a condition which implies that $[\![\mathcal{S}(p,q)]\!]$ itself is a safe zone? As a matter of fact, yes!

Definition 3.1.14 (Quasi-discrete closure spaces)**.** Given a closure space $\langle X, \mathcal{C}\rangle$ we say it is *quasi-discrete* if it holds:

$$\mathcal{C}(A) = \bigcup_{s \in A} \mathcal{C}(\{s\})$$

Remark 3.1.15. Note that the space in the Example 3.1.13 *is not* quasi-discrete, so not all closure spaces are quasi-discrete.

First of all, let's check that in a quasi-discrete space $[\![\mathcal{S}(p,q)]\!]$ is a safe zone (in the sense mentioned above).

Lemma 3.1.16. *Let $\mathcal{X} = \langle X, \mathcal{C}, g\rangle$ be a quasi-discrete space. Then it holds:*

$$[\![\mathcal{S}(p,q)]\!] \subseteq [\![p]\!]$$
$$\mathcal{B}^+([\![\mathcal{S}(p,q)]\!]) \subseteq [\![q]\!]$$

Proof. The first part is trivial. For the second we have:

$$\begin{aligned}
\mathcal{C}([\![\mathcal{S}(p,q)]\!]) &= \bigcup_{s \in [\![\mathcal{S}(p,q)]\!]} \mathcal{C}(\{s\}) \\
&\Downarrow \\
\mathcal{B}^+([\![\mathcal{S}(p,q)]\!]) &= \bigcup_{s \in [\![\mathcal{S}(p,q)]\!]} (\mathcal{C}(\{s\}) \setminus [\![\mathcal{S}(p,q)]\!]) \\
&\subseteq \bigcup_{s \in [\![\mathcal{S}(p,q)]\!]} (\mathcal{C}(\{s\}) \setminus \{s\}) \\
&= \bigcup_{s \in [\![\mathcal{S}(p,q)]\!]} \mathcal{B}^+(\{s\}) \subseteq [\![q]\!]
\end{aligned}$$

□

Note that quasi-discrete spaces have also some other interesting properties:

Lemma 3.1.17.

1. *In a quasi-discrete space the closure operator is completely determined by its value on singletons. Formally, given a set X and a function $f : X \to \mathcal{P}(X)$ such that for all $s \in X$ it holds $s \in f(s)$, there exists one and only one closure operator $\mathcal{C}$ over X such that*

$$\mathcal{C}(\{s\}) = f(s)$$

   and $\langle X, \mathcal{C}\rangle$ is a quasi-discrete space.

2. *Every finite closure space is quasi-discrete.*

What we want to show now is that quasi-discrete spaces and `KFs` are *essentially the same thing* when we deal with the semantics of `SLCS`.

Notation 3.1.18.

- We define $\texttt{KF}^{\circlearrowleft}$ as the class of `KFs` such that every state has a self loop. Formally, $\mathcal{M} = \langle S, \to, L\rangle$ is in $\texttt{KF}^{\circlearrowleft}$ if

$$\forall s \in S.\, s \to s$$

- We define `qDiscrete` as the class of quasi-discrete closure spaces with environment.

Theorem 3.1.19. *Given $\mathcal{M} = \langle S, \to, L\rangle$ a $\texttt{KF}^{\circlearrowleft}$ we can define a quasi-discrete closure operator $\mathcal{C}$ such that:*

$$t \in \mathcal{C}(\{s\}) \iff s \to t$$

Moreover, if we define the class function $F : \texttt{KF}^{\circlearrowleft} \to \texttt{qDiscrete}$ such that $F(\mathcal{M}) = \langle S, \mathcal{C}, L\rangle$ then F is a 1:1 correspondence between $\texttt{KF}^{\circlearrowleft}$ and `qDiscrete`.

Corollary 3.1.20. *The restriction of F to finite $\texttt{KF}^{\circlearrowleft}$ is an invertible class function with image the finite quasi-discrete spaces.*

Now that we established this, we can define a semantics for `SLCS` whose models are `KFs` with self loops by *transferring* the previous semantics. What we obtain is the following:

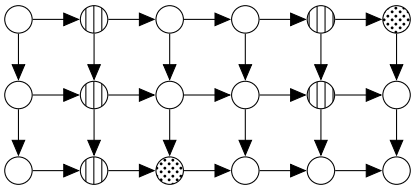
Definition 3.1.21 (`KF` semantics for `SLCS`)**.** Given a $\texttt{KF}^{\circlearrowleft}$ $\mathcal{X} = \langle X, \to, L\rangle$, given $s \in X$, given an `SLCS` formula φ, we want to define the entailment

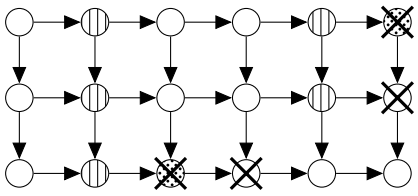
relation $\mathcal{X}, s \vDash \varphi$. As in definition (3.1.9) we will denote with $[\![\varphi]\!]$ the set of elements $t \in X$ for which $\mathcal{X}, t \vDash \varphi$ holds.

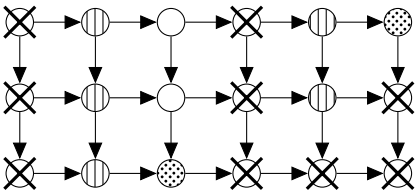
$$
\begin{aligned}
&\mathcal{X}, s \nvDash \bot \\
&\mathcal{X}, s \vDash p \in \texttt{AP} && \iff s \in L(p) \\
&\mathcal{X}, s \vDash \varphi_1 \wedge \varphi_2 && \iff \mathcal{X}, s \vDash \varphi_1 \text{ and } \mathcal{X}, s \vDash \varphi_2 \\
&\mathcal{X}, s \vDash \neg\varphi && \iff \mathcal{X}, s \nvDash \varphi \\
&\mathcal{X}, s \vDash \mathcal{S}(\varphi_1, \varphi_2) && \iff \text{there exists } A \subseteq X \text{ such that } s \in A \\
& && \qquad \text{and } \begin{cases} A & \subseteq & [\![\varphi_1]\!] \\ \mathrm{FN}(A) \setminus A & \subseteq & [\![\varphi_2]\!] \end{cases} \\
&\mathcal{X}, s \vDash \mathcal{N}(\varphi) && \iff \text{there exists } t \in [\![\varphi]\!] \text{ such that } t \to s
\end{aligned}
$$

where $\mathrm{FN}(A) = \bigcup_{a \in A} \mathrm{FN}(a)$.

Example 3.1.22. Consider the $\texttt{KF}^{\circlearrowleft}$ represented in figure, where the self loops are omitted and the atomic propositions are p (dotted nodes), q (white nodes) and r (nodes with vertical lines). On the left, the semantics of the formula $\mathcal{N}(p)$. On the right the semantics of the formula $\mathcal{S}(q, r)$.







We don't present here the algorithms to solve the SLCS model checking problem, as in the next section we will solve it for STLCS, a generalization of the logic SLCS. The original algorithms can be found in [15] and [5].

However, the correctness of the algorithms is based on a Lemma that we state and prove here and we will generalize in the next sections.

Definition 3.1.23 ($(\varphi, \psi)^+$-path)**.** We define the finite path π to be a $(\varphi, \psi)^+$-path if the followings hold:

- $l = \mathrm{length}(\pi) \geq 1$.
- For all k such that $0 < k < l$ it holds $\pi_k \vDash \varphi$.

- $\pi_l \vDash \psi$.

We indicate the set of $(\varphi, \psi)^+$-paths starting at s with $\Pi_s^+(\varphi, \psi)$.

Lemma 3.1.24.

$$s \vDash \mathcal{S}(p,q) \iff s \vDash p \text{ and } \Pi_s^+(\neg q, \neg p \wedge \neg q) = \emptyset$$

Proof.

left-to-right implication: Suppose $s \vDash \mathcal{S}(p,q)$, then clearly $s \vDash p$. To prove the other conjunct consider S a set of states such that

$$\begin{array}{rcl} S & \subseteq & [\![p]\!] \\ \mathcal{B}^+(S) & \subseteq & [\![q]\!] \end{array}$$

and a finite path π starting at s. Then we have two cases:

- Every state of π is in S, and so

$$\forall i \in \mathbb{N}.\pi_i \in [\![p]\!] \implies \pi \notin \Pi_s^+(\neg q, \neg p \wedge \neg q)$$

- There exists a least $l \in \mathbb{N}$ such that $\pi_l \notin S$, and so $l \geq 1$ and $\pi_l \in \mathcal{B}^+(S) \subseteq [\![q]\!]$. But this entails that $\pi \notin \Pi_s^+(\neg q, \neg q \wedge \neg p)$ as each state of a $(\neg q, \neg q \wedge \neg p)^+$-path (except the first one) has to entail $\neg q$.

right-to-left implication: consider the set S defined as

$$S = \left\{ t \;\middle|\; \begin{array}{l} t \text{ is reachable from } s \text{ with a path } \pi = (s = \pi_0, \ldots, \pi_l) \\ \text{such that } \pi_1, \ldots, \pi_l \in [\![\neg q]\!] \end{array} \right\}$$

From the property $\Pi_s^+(\neg q, \neg q \wedge \neg p) = \emptyset$ it follows that each $t \in S$ entails p. The claim is that S is a "good set" to prove the thesis, meaning that $S \subseteq [\![p]\!]$ and $\mathcal{B}^+(S) \subseteq [\![q]\!]$.

By contradiction, suppose there exists $u \in \mathcal{B}^+(S)$ such that $u \vDash \neg q$. Then we have:

$$\begin{array}{rcr} u \in \mathcal{B}^+(S) & \implies & \text{there exists } t \in S \text{ such that } t \to u \\ & \implies & \text{there exists a path from } s \text{ to } u \text{ in } [\![\neg q]\!] \\ & \implies & u \in S \end{array}$$

but this is absurd as $u \in \mathcal{B}^+(S)$. Now it's easy to prove that $S' = S \cup \{s\}$ has exactly the properties required to prove $s \vDash \mathcal{S}(p,q)$.

□

3.2 Spatio-Temporal Logic of Closure Spaces

Definition 3.2.1 (Syntax of STLCS)**.** The syntax of STLCS formulas is defined by the following grammar:

$$\begin{array}{lll} \langle\varphi\rangle & \models & \bot \mid p \mid \langle\varphi\rangle\wedge\langle\varphi\rangle \mid \neg\langle\varphi\rangle \mid \mathtt{E}\langle\Phi\rangle \mid \mathtt{A}\langle\Phi\rangle \mid \mathcal{S}(\langle\varphi\rangle,\langle\varphi\rangle) \mid \mathcal{N}(\langle\varphi\rangle) \\ \langle\Phi\rangle & \models & \mathtt{X}\langle\varphi\rangle \mid \mathtt{F}\langle\varphi\rangle \mid \mathtt{G}\langle\varphi\rangle \mid \langle\varphi\rangle\mathtt{U}\langle\varphi\rangle \end{array}$$

where $p \in \mathtt{AP}$.

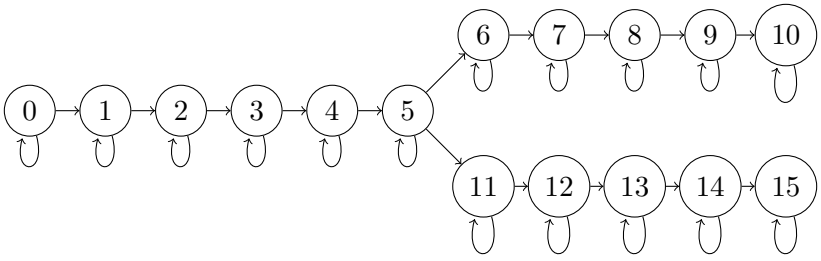
We will call formulas of the first sort *state formulas* and formulas of the second sort *time-path formulas* (or simply path formulas).

Moral 3.2.2. It's easy to notice that this syntax resembles both the syntax of CTL (Definition 1.1.1) and the syntax of SLCS (Definition 3.1.1). In fact, as the name suggests, the idea behind this logic is to describe the spatio-temporal structure of a system by interleaving spatial properties (i.e., properties definable by SLCS operators) and temporal properties (i.e., properties definable by CTL operators).

Definition 3.2.3 (STLCS model)**.**

- Given $\mathcal{X} = \langle X, \mathcal{C}\rangle$ a closure space and $\mathcal{S} = \langle S, \to\rangle$ a transition system, an *environment* over $\langle\mathcal{X}, \mathcal{S}\rangle$ is a function $g : \mathtt{AP} \to \mathcal{P}(X \times S)$.
- We define a *spatio-temporal model* (or simply a *model*) for STLCS to be a tuple $\mathcal{M} = \langle\mathcal{X}, \mathcal{S}, g\rangle$ where $\mathcal{X}$ is a closure space, $\mathcal{S}$ is a transition system and g is an environment over $\langle\mathcal{X}, \mathcal{S}\rangle$.

Example 3.2.4. By considering an image as a lattice graph, we can define a spatio-temporal model by giving a Kripke frame and a sequence of images. As an example consider the Kripke frame in Figure 3.2.4 and the sequence of images in Figure 3.2.4.



Notation 3.2.5. Fix $\mathcal{M}$ a spatio-temporal model. Referring to the notations above:

- We will call an element of $X \times S$ a *state*.

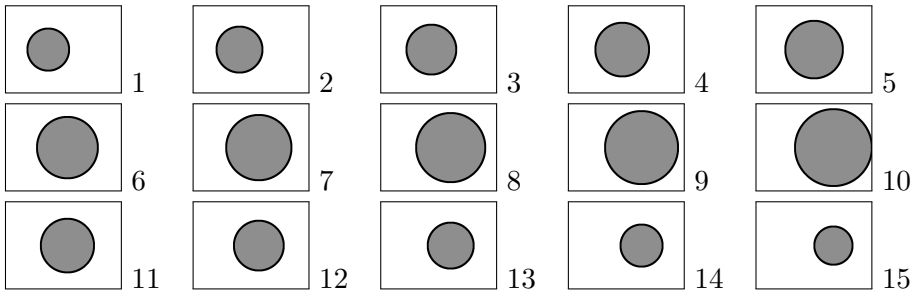


Figure 3.6: We consider two atomic properties (black and grey), whose interpretation in the model are encoded by the images.

- We will call an element of X a *space state* and an element of S a *time state*.
- We will call a path in X a *space path* and a path of S a *time path*.
- Fix $A \subseteq X \times S$, $x \in X$ and $s \in S$. Then we define the sections

$$A^x = \{s \in S | \langle x, s\rangle \in A\}$$
$$A_s = \{x \in X | \langle x, s\rangle \in A\}$$

Now we proceed to give the semantics for the logic.

Definition 3.2.6 (Spatio-temporal model semantics for `STLCS`)**.** Given $\mathcal{X} = \langle X, \mathcal{C}\rangle$ a closure space, $\mathcal{S} = \langle S, \to\rangle$ a transition system and $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g\rangle$ a spatio temporal model, we want to define what it means that a state $\langle x, s\rangle \in X \times S$ *entails* a state formula φ and that a pair $\langle x, \pi\rangle$ (with x a state space and π a time-path) *entails* a path formula Φ (in symbols $\mathcal{M}, x, s \vDash \varphi$ and $\mathcal{M}, \pi \vDash \Phi$ respectively). To simplify the next definition we introduce the following notations:

- $[\![\varphi]\!] \subseteq X \times S$ is the set of states which entails φ (so the pairs $\langle x, s\rangle$ such that $\mathcal{M}, x, s \vDash \varphi$).
- $[\![\varphi]\!]^x \subseteq S$ is the set of time states s such that $\mathcal{M}, x, s \vDash \varphi$ (in accord to the definition of section).
- $[\![\varphi]\!]_s \subseteq X$ is the set of space states x such that $\mathcal{M}, x, s \vDash \varphi$ (in accord to the definition of section).

By induction over the formula φ we define:

$$
\begin{array}{ll}
\mathcal{M}, x, s \not\models \bot & \\
\mathcal{M}, x, s \models p \in \mathtt{AP} & \iff \langle x, s\rangle \in g(p) \\
\mathcal{M}, x, s \models \varphi_1 \wedge \varphi_2 & \iff \mathcal{M}, x, s \models \varphi_1 \text{ and } \mathcal{M}, x, s \models \varphi_2 \\
\mathcal{M}, x, s \models \neg\varphi & \iff \mathcal{M}, x, s \not\models \varphi \\
\mathcal{M}, x, s \models \mathtt{E}\Phi & \iff \text{exists a time-path } \pi \text{ with } \pi_0 = s \\
 & \quad \text{such that } \mathcal{M}, x, \pi \models \Phi \\
\mathcal{M}, x, s \models \mathtt{A}\Phi & \iff \text{for all time-paths } \pi \text{ such that} \\
 & \quad \pi_0 = s \text{ it holds } \mathcal{M}, x, \pi \models \Phi \\
 & \\
\mathcal{M}, x, \pi \models \mathtt{X}\varphi & \iff \mathcal{M}, x, \pi_1 \models \varphi \\
\mathcal{M}, x, \pi \models \mathtt{F}\varphi & \iff \text{there exists } n \geq 0 \text{ such that } \mathcal{M}, x, \pi_n \models \varphi \\
\mathcal{M}, x, \pi \models \mathtt{G}\varphi & \iff \text{for all } n \geq 0 \text{ it holds } \mathcal{M}, x, \pi_n \models \varphi \\
\mathcal{M}, x, \pi \models \varphi_1 \mathtt{U} \varphi_2 & \iff \text{there exists } n \geq 0 \text{ such that } \mathcal{M}, x, \pi_n \models \varphi_2 \\
 & \quad \text{and for all } k \leq n \ \mathcal{M}, x, \pi_k \models \varphi_1 \\
\mathcal{M}, x, s \models \mathcal{S}(\varphi_1, \varphi_2) & \iff \text{there exists } A \subseteq X \text{ such that } x \in A \\
 & \quad \text{and } \begin{cases} A & \subseteq & [\![\varphi_1]\!]_s \\ \mathcal{B}^+(A) & \subseteq & [\![\varphi_2]\!]_s \end{cases} \\
\mathcal{M}, x, s \models \mathcal{N}(\varphi) & \iff x \in \mathcal{C}\left([\![\varphi]\!]_s\right)
\end{array}
$$

Moral 3.2.7. The idea behind this inductive definition is that a spatial operator describes only properties of the spatial component of the model, while time operators describe only properties of the time component. This statement can be formalized with the next results.

Definition 3.2.8. Let φ be a STLCS formula and $\psi_1, \ldots, \psi_k$ be it's direct sub-formulas. We say that φ is:

- *a local formula* if for every state $\langle x, s\rangle$ of every model, the truth value of φ at $\langle x, s\rangle$ depends only on the truth value of $\psi_1, \ldots, \psi_k$ at $\langle x, s\rangle$.
- *a spatial formula* if for every time state s of every model $\langle \mathcal{X}, \mathcal{S}, g\rangle$, the value of $[\![\varphi]\!]_s$ depends only on $\mathcal{X}$ and the function $g(\bullet)_s$.
- *a temporal formula* if for every space state x of every model $\langle \mathcal{X}, \mathcal{S}, g\rangle$, the value of $[\![\varphi]\!]^x$ depends only on $\mathcal{S}$ and the function $g(\bullet)^x$.

Lemma 3.2.9. *Fix $\mathcal{M}$ a spatio-temporal model (we will use the notations above). Then:*

- *If φ and ψ are local formulas, then so are $\varphi \wedge \psi$ and $\neg\varphi$.*

- *If φ and ψ are temporal formulas, then so are $\mathtt{EX}(\varphi)$, $\mathtt{EF}(\varphi)$, $\mathtt{EG}(\varphi)$, $\mathtt{E}(\varphi \mathtt{U} \psi)$, $\mathtt{AX}(\varphi)$, $\mathtt{AF}(\varphi)$, $\mathtt{AG}(\varphi)$ and $\mathtt{A}(\varphi \mathtt{U} \psi)$.*
- *If φ and ψ are spatial formulas, then so are $\mathcal{N}(\varphi)$ and $\mathcal{S}(\varphi, \psi)$.*

Corollary 3.2.10.

- *Consider φ a CTL formula as a STLCS formula in the natural way. Then φ is a temporal formula.*
- *Consider φ a SLCS formula as a STLCS formula in the natural way. Then φ is a spatial formula.*

A finer result is the following:

Lemma 3.2.11. *Fix a model $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g \rangle$. Then:*

- *Given φ a CTL formula and $x \in X$, it holds:*

$$[\![\varphi]\!]^x = [\![\varphi]\!]^{CTL}_{\langle \mathcal{S}, g(\bullet)^x \rangle}$$

  where $[\![\varphi]\!]^{CTL}_{\langle \mathcal{S}, g(\bullet)^x \rangle}$ indicates the CTL semantics of φ considering $\langle \mathcal{S}, g(\bullet)^x \rangle$ as a KF.

- *Given φ a SLCS formula and $s \in S$, it holds:*

$$[\![\varphi]\!]_s = [\![\varphi]\!]^{SLCS}_{\langle \mathcal{X}, g(\bullet)_s \rangle}$$

  where $[\![\varphi]\!]^{SLCS}_{\langle \mathcal{X}, g(\bullet)_s \rangle}$ indicates the SLCS semantics of φ considering $\langle \mathcal{X}, g(\bullet)_s \rangle$ as a closure space.

These results show us that this logic has both the expressive power of CTL and SLCS, but maintains the two interpretations well separated. To further strengthen this assertion, we present the following result.

Lemma 3.2.12. *Consider φ an STLCS formula that is both spatial and temporal. Then φ is local.*

Proof. Recalling the definitions above, we have that:

- φ is a spatial formula means that, fixed a closure space $\mathcal{X}$ and a function $f : \mathtt{AP} \to \mathcal{P}(X)$, then for every spatio-temporal model $\langle \mathcal{X}, \mathcal{S}, g \rangle$, for every time state s such that $g(\bullet)_s = f$, the value $[\![\varphi]\!]_s$ is the same.
- φ is a temporal formula means that, fixed a transition system $\mathcal{S}$ and a function $L : \mathtt{AP} \to \mathcal{P}(S)$, then for every spatio-temporal model $\langle \mathcal{X}, \mathcal{S}, g \rangle$, for every space state x such that $g(\bullet)^x = L$, the value $[\![\varphi]\!]^x$ is the same.

Now we consider an arbitrary spatio-temporal model $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g \rangle$ and we want to find the truth value of φ at a state $\langle x, s \rangle$.

If we define

- $\mathcal{S}^* =$  , the transition system with a single state $*$ and a single self-loop.
- $g' : \mathtt{AP} \to \mathcal{P}(X \times \{*\})$ such that $g' = g(\bullet)_s \times \{*\}$.

then by the first point above, defining $\mathcal{M}' = \langle \mathcal{X}, \mathcal{S}^*, g' \rangle$ we have that

$$\mathcal{M}', x, * \vDash \varphi \iff \mathcal{M}, x, s \vDash \varphi$$

Using the same trick we can define

- $\mathcal{X}^* =$  (modulo the identification of finite closure spaces and graphs introduced before).
- $g'' : \mathtt{AP} \to \mathcal{P}(\{*\} \times \{*\})$ such that $g'' = g'(\bullet)^x$. So in particular

$$g''(p) = \begin{cases} \{*\} \times \{*\} & \text{if } \langle x, s \rangle \in g(p) \\ \emptyset & \text{otherwise} \end{cases}$$

And now by the second point above, we have that defining $\mathcal{M}'' = \langle \mathcal{X}^*, \mathcal{S}^*, g'' \rangle$ it holds:

$$\mathcal{M}'', *, * \vDash \varphi \iff \mathcal{M}', x, * \vDash \varphi \iff \mathcal{M}, x, s \vDash \varphi$$

But now we have that the model $\mathcal{M}''$ depends only on the set $\{p \in \mathtt{AP} | \langle x, s \rangle \in g(p)\}$, and so φ is local.

□

Finally we present here the algorithms to compute the semantics of the formulas. We premise a technical lemma:

Lemma 3.2.13 (Temporal E Normal Form)**.** *Let φ be a STLCS formula. Then there exists a formula φ^E produced by the grammar*

$$\begin{aligned} \langle \varphi \rangle \quad \models \quad & \bot \mid p \in \mathtt{AP} \mid \langle \varphi \rangle \wedge \langle \varphi \rangle \mid \neg \langle \varphi \rangle \mid \\ & \mathtt{EX}\langle \varphi \rangle \mid \mathtt{EG}\langle \varphi \rangle \mid \mathtt{E}\langle \varphi \rangle \mathtt{U} \langle \varphi \rangle \mid \\ & \mathcal{N}\langle \varphi \rangle \mid \mathcal{S}(\langle \varphi \rangle ; \langle \varphi \rangle) \end{aligned}$$

such that for every spatio-temporal model it holds:

$$[\![\varphi]\!] = [\![\varphi^E]\!]$$

The proof follows the exactly same argument as Lemma 1.1.8.

Thanks to this lemma we only need to give algorithms to compute the semantics for the formulas $\mathtt{EX}(p)$, $\mathtt{EG}(p)$, $\mathtt{E}(p \mathtt{U} q)$, $\mathcal{N}(p)$ and $\mathcal{S}(p, q)$, as we can compute the semantics of a formula φ by induction over the structure of $\varphi^{\mathtt{E}}$. The Algorithms are 7, 8, 9, 10 and 11 below.

Algorithm 7: Algorithm to compute $\texttt{EX}p$

input : $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g \rangle$, $\texttt{EX}p$
output: $[\![\texttt{EX}p]\!]$

```
1 semP:= [[p]];
2 sectionsDictionary:= {(x, semP^x) | x ∈ X};
3 res:= ∅;
4 for x ∈ X do
5     for s ∈ sectionsDictionary[x] do
6         predSet:= PredTimeStates(s);
7         res:= Union({x}× predSet, res);
  return : res
```

Algorithm 8: Algorithm to compute $\texttt{EG}p$

input : $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g \rangle$, $\texttt{EG}p$
output: $[\![\texttt{EG}p]\!]$

```
1 SemP:= [[p]];
2 SectionsDictionary:= {(x, res^x) | x ∈ X};
3 for x ∈ X do
4     semNotPAtX:= S \ SectionsDictionary[x];
5     corrosionLayer:= Intersection(predSet(semNotPAtX),
        SectionsDictionary[x]);
6     while corrosionLayer ! = ∅ do
7         SectionsDictionary[x] := SectionsDictionary[x] \ corrosionLayer;
8         corrosionLayer:=
            Intersection(PredTimeStates(corrosionLayer),
            SectionsDictionary[x]);
9 res:= ⋃_{x∈X} ({x} × SectionsDictionary[x]);
  return : res
```

Algorithm 9: Algorithm to compute $\mathtt{E}(p\mathtt{U}q)$

input : $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g \rangle$, $\mathtt{E}(p\mathtt{U}q)$
output: $[\![\mathtt{E}(p\mathtt{U}q)]\!]$

couldBe:= $[\![p]\!]$;
SemQ:= $[\![q]\!]$;
SectionsCouldBe:= $\{(x, \mathsf{couldBe}^x) | x \ inX\}$;
SectionsRes:= $\{(x, \mathsf{res}^x) | x \in X\}$;
for $x \in X$ **do**
    toAdd:= `Intersection(predSet(`SectionsRes$[x]$`),` SectionsCouldBe$[x]$`);`
    SectionsCouldBe$[x]$:= SectionsCouldBe$[x] \setminus$ toAdd;
    while toAdd $!= \emptyset$ **do**
        SectionsCouldBe$[x]$:= `Union(`SectionsRes$[x]$, toAdd`);`
        toAdd:= `Intersection(predSet(`toAdd`),` SectionsCouldBe$[x]$`);`
        SectionsCouldBe$[x]$:= SectionsCouldBe$[x] \setminus$ toAdd;
res:= $\bigcup_{x \in X} (\{x\} \times$ SectionsRes$[x])$;
return : res

Algorithm 10: Algorithm to compute $\mathcal{N}\, p$

input : $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g \rangle$, $\mathcal{N}\, p$
output: $[\![\mathcal{N}\, p]\!]$

semP:= $[\![p]\!]$;
sectionsDictionary:= $\{(\mathsf{semP}_s, s) | s \in S\}$;
res:= $\emptyset$;
for $s \in S$ **do**
    for $x \in$ sectionsDictionary$[s]$ **do**
        SuccSet:= `SuccSpaceStates(`s`);`
        res:= (SuccSet $\times \{s\}) \cup$ res;
return : res

Lemma 3.2.14. *The algorithms presented terminate correctly in time linear in the encoding of the model.*

Proof idea. The proof of this result is a direct consequence of Lemma 3.2.9.

In fact the algorithms first compute a division in sections of the sets involved (for example, in the algorithm to compute $\mathtt{EX}p$ the division is stored in the array sectionsDictionary in line 2) and then apply a suitable algorithm to compute the semantics of the formula at each section.

The algorithms used to compute the semantics of the sections are 1, 2, 3 (in the temporal case) and the algorithms presented in [15] and [5].

As the computation of the sections and each algorithm involved have linear-time complexity, so is the whole model-checking procedure. □

Algorithm 11: Algorithm to compute $\mathcal{S}(p, q)$

input : $\mathcal{M} = \langle \mathcal{X}, \mathcal{S}, g \rangle$, $\mathcal{S}(p, q)$
output: $[\![\mathcal{S}(p, q)]\!]$

```
1  SemP:= [[p]];
2  SemQ:= [[q]];
3  SectionsSemSPQ:= {(SemP_s, s)|s ∈ S};
4  SectionsSemQ:= {(SemQ_s, s)|s ∈ S};
5  for s ∈ S do
6      BadPoints:= S \ (SectionsSemSPQ[s] ∪ SectionsSemQ[s]);
7      for bp ∈ BadPoints do
8          BadPoints:= BadPoints \ bp;
9          ToRemove:=
             PredSpaceStates( { bp } ) ∩ SectionsSemSPQ[s];
10         SectionsSemSPQ[s] := SectionsSemSPQ[s] \ ToRemove;
11         BadPoints := BadPoints ∪ (ToRemove \ SectionsSemQ[s]);
12 res:= ⋃_{s∈S} (SectionsSemSPQ × {s});
   return : res
```

Chapter 4

Symbolic Model Checking and Abstraction

In this chapter we present for the first time two methods developed by the author to solve more efficiently the model checking problem for the logic `SLCS`. In particular, they are an adaptation of the methods introduced in Chapter 2, namely symbolic model checking and abstraction-refinement. The aim of these methods is to speed up the model checking procedure when the model has a strong structure not directly encoded by the `KF`, so for example when it's obtained from the analysis of a *series of images*. These methods have not yet been implemented and so a thorough analysis will be presented in future works.

For the symbolic model checking procedure, the algorithm is a simple adaptation of the one presented in Chapter 2, but the benefit of adapting it is considerable. In fact the symbolic model checking algorithm proves to be really fast when dealing with models representing *parallel systems*, and so the natural claim is that when dealing with *product spaces* a similar speed up is achieved.

On the other hand, for the abstraction-refinement method the algorithm can't be trivially adapted to the spatial case. The main reason is that, in the case of a spatial model representing a real-life system there usually is a hierarchical structure (for example the plant of a building is subdivided in floors, the floors in zones, ...) that is usually lost when considering a quotient as the bisimulation one presented in Chapter 2. Preserving and using the implicit hierarchical structure could give a huge improvement in the speed of the procedure.

A new abstraction-refinement method is presented here, which preserve a fixed hierarchical structure provided together with the model. The main conceptual differences with the procedures introduced in Chapter 2 are two, namely the introduction of *nested abstractions* (allowing for a gradual refinement of the model) and the method to *choose the states to expand*.

Of course, this algorithm is thought for models which represent systems with a hierarchical structure, so in the general case a substantial speed-up is not guaranteed. However in the strongly structured case an improvement in the computation time is expected.

4.1 Symbolic Model Checking for SLCS

In this chapter we will show that the symbolic model checking algorithm can be adapted to solve the MCP for the logic SLCS. To do so, we only need to write the model checking algorithm presented above in a symbolic way, and then to check which operations must be computed efficiently, exactly as in the CTL case.

We begin by writing the symbolic algorithms and by proving they are correct. We use here the same notation introduced in Algorithms 4, 5 and 6.

Algorithm 12: Symbolic algorithm to compute $\mathcal{N}(p)$

input : f_p, the switching function corresponding to $[\![p]\!]$
output: $f_{\mathcal{N}(p)}$, the switching function corresponding to $[\![\mathcal{N}(p)]\!]$

1 $\mathsf{g}(\overline{x}) := \exists \overline{x}'.\,(\Delta(\overline{x}', \overline{x}) \wedge f_p(\overline{x}'))$;
2 **return** g;

Algorithm 13: Symbolic algorithm to compute $\mathcal{S}(p, q)$

input : f_p, the switching function corresponding to $[\![p]\!]$
input : f_q, the switching function corresponding to $[\![q]\!]$
output: $f_{\mathcal{S}(p,q)}$, the switching function corresponding to $[\![\mathcal{S}(p, q)]\!]$

1 $f_0 := f_p$;
2 j:= 0;
3 **repeat**
4 $\quad \mathsf{g}(\overline{x}, \overline{x}') := \Delta(\overline{x}, \overline{x}') \rightarrow (f_j(\overline{x}') \vee f_q(\overline{x}'))$;
5 $\quad f_{j+1}(\overline{x}) := f_j(\overline{x}) \wedge \forall \overline{x}'.\mathsf{g}(\overline{x}, \overline{x}')$;
6 $\quad$ j:= j +1;
7 **until** $f_j == f_{j-1}$;
8 **return** f_j;

Theorem 4.1.1. *Algorithms 12 and 13 are correct and terminate in a finite number of steps.*

Proof. In the following proof we fix a model $\mathcal{M} = \langle S, \rightarrow, L \rangle$. Moreover we suppose to have already encoded the model for the symbolic algorithm, namely that

- $S = \{0, \ldots, 2^n - 1\}$
- $\rightarrow$ is represented by the switching function $\Delta : \{0, 1\}^{2n} \rightarrow \{0, 1\}$

- Each L_p is represented by the switching function $f_p : \{0,1\}^n \to \{0,1\}$

Case $\mathcal{N}$: The algorithm terminates since it doesn't contain loops. Now we have to check that the function returned by the algorithm ($f_{\mathcal{N}(p)}$) has the property we want, namely:

$$f_{\mathcal{N}(p)}(\overline{x}) = 1 \Leftrightarrow \overline{x} \in [\![\mathcal{N}\, p]\!]$$

but now it follows directly from the code that

$$\begin{aligned} f_{\mathcal{N}(p)}(\overline{x}) = 1 &\Leftrightarrow \text{there exists } \overline{x}' \text{ such that } \Delta(\overline{x}', \overline{x}) = 1 \text{ and } f_p(\overline{x}') = 1 \\ &\Leftrightarrow \text{there exists } \overline{x}' \text{ such that } \overline{x}' \to \overline{x} \text{ and } \overline{x}' \in [\![p]\!] \\ &\Leftrightarrow \overline{x} \in [\![\mathcal{N}(p)]\!] \end{aligned}$$

as wanted.

Case $\mathcal{S}$: First of all, we show that the succession of functions $(f_j)_{j\in\mathbb{N}}$ generated by the algorithm is (point-wise) decreasing and so, as the model is finite, it reaches a fixed point. This proves that the algorithm terminates, as the only loop terminates if $f_j = f_{j-1}$.

In line 5 we have

$$f_{j+1}(\overline{x}) = 1 \iff f_j(\overline{x}) = 1 \text{ and other conditions} \Rightarrow f_j(\overline{x}) = 1$$

and so the chain of functions is decreasing as wanted. Note that this proves also that this chain stabilizes in almost $2^n = |S|$ steps, so $f_{\mathcal{S}(p,q)} = f_{2^n-1}$.

Now we have to check that the algorithm is correct, namely that for the output function $f_{\mathcal{S}(p,q)}$ it holds:

$$f_{2^n-1}(\overline{x}) = f_{\mathcal{S}(p,q)}(\overline{x}) = 1 \iff \overline{x} \in [\![\mathcal{S}(p,q)]\!]$$

To prove this, we give a characterization of the functions f_j:

Claim 4.1.2. *$f_j(\overline{x}) = 0$ if and only if $f_p(\overline{x}) = 0$ or there exists a path $\pi \in \Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q)$ of length at most j.*

The moral behind this result, is that f_j check if there is a *counterexample* to the truth of $\mathcal{S}(p,q)$ of bounded length.

Given this result, the thesis follows from Lemma 3.1.24.

□

Proof of claim 4.1.2. We prove the result by induction on $j \in \mathbb{N}$.

If $j = 0$ the result is trivially true. So let's suppose $j > 0$. By definition of f_j we have:

$$\begin{aligned} f_j(\overline{x}) = 0 &\iff f_{j-1}(\overline{x}) \wedge \forall \overline{x}'.\, \big(\Delta(\overline{x}, \overline{x}') \to (f_{j-1}(\overline{x}') \vee f_q(\overline{x}'))\big) = 0 \\ &\iff f_{j-1}(\overline{x}) = 0 \text{ or} \\ &\qquad \text{exists } \overline{x}' \in \mathrm{FN}(\overline{x}) \text{ such that } f_{j-1}(\overline{x}') \vee f_q(\overline{x}') = 0 \end{aligned}$$

Suppose now that $f_j(\overline{x}) = 0$ and consider the two clauses of the last disjunction. We have that:

- if $f_{j-1}(\overline{x}) = 0$. Then by inductive hypothesis $f_p(\overline{x}) = 0$ or there exists a path $\pi \in \Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q)$ of length at most $j - 1$, as wanted.
- if there exists $\overline{x}' \in \mathrm{FN}(\overline{x})$ such that $f_{j-1}(\overline{x}') = 0$ and $f_q(\overline{x}') = 0$. Then again, by inductive hypothesis, the condition $f_{j-1}(\overline{x}') = 0$ means one of the following cases applies:
  - $f_p(\overline{x}') = 0$**:** it follows that $\overline{x}' \in [\![\neg p \wedge \neg q]\!]$. If we consider the path $(\overline{x} \to \overline{x'})$ we have the thesis as this path is in $\Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q)$.
  - **there exists a path** $\pi \in \Pi^+_{\overline{x}'}(\neg q, \neg p \wedge \neg q)$ **of length** $l \leq j - 1$**:** in this case we can consider the path

    $$\pi' = \overline{x} \frown \pi = (\overline{x} \to \pi_0 \to \pi_1 \to \cdots \to \pi_{l-1})$$

    and this is a path in $\Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q)$, as:
    - $0 < \mathrm{length}(\pi') = \mathrm{length}(\pi) + 1 \leq j$
    - For all $0 < i < \mathrm{length}(\pi')$ it holds $\pi'_i \in [\![\neg q]\!]$, as for $i > 1$ it follows from the properties of π and for $i = 1$ it follows from $\pi'_1 = \pi_0 = \overline{x}'$.
    - $\pi'_l = \pi_{l-1} \in [\![\neg p \wedge \neg q]\!]$

    In both cases we found a $(\neg q, \neg p \wedge \neg q)^+$-path, and so this proves $\Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q) \neq \emptyset$, as wanted.

This proves the first implication, namely that $f_j(\overline{x}) = 0$ implies the existence of a path in $\Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q)$ or that $f_p(\overline{x}) = 0$.

For the other implication, suppose at first that $f_p(\overline{x}) = 0$. Then by monotonicity of the succession f_j and the fact that $f_0 = f_p$, we have that $f_j(\overline{x}) = 0$. So we only have to consider the case $f_p(\overline{x}) \neq 0$, that is $\Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q) \neq \emptyset$.

Let $\pi \in \Pi^+_{\overline{x}}(\neg q, \neg p \wedge \neg q)$ and let $l > 0$ be its length. Since $l > 0$, we can define $\overline{x}' := \pi_1$ and it's easy to show that $f_{l-1}(\overline{x}') = 0$ and $f_q(\overline{x}') = 0$:

- $f_q(\overline{x}') = 0$ follows directly from the properties of π.

- To prove that $f_{l-1}(\overline{x}') = 0$ we proceed by induction on length(π):
  - If length(π) $= 1$ then $\overline{x}' \in [\![\neg p \wedge \neg q]\!] \subseteq [\![\neg q]\!]$.
  - If length(π) > 1 then $\pi' = (\pi_1 \to \cdots \to \pi_l)$ (obtained by removing the head to the succession π) is a path in $\Pi^{+}_{x'}(\neg q, \neg p \wedge \neg q)$ of length $l - 1$. So by inductive hypothesis $f_{l-2}(\pi_2) = 0$, but know it's easy to show that $f_{l-1}(\pi_1) = 0$, as wanted (lines 4-5).

From this we conclude that also the other implication holds. □

Now that we proved the correctness of the model checking algorithm, we need to give an efficient method to compute the basic operations inside the code. But note that:

- We already studied an efficient data structure to compute operators $\wedge, \vee, \exists$ using switching function, namely `OBDDs`.
- Given f and g switching functions, it holds $[f \implies g] \equiv [\neg f \vee g]$
- Given $h(\overline{x}, \overline{x}')$ a switching function, it holds $\forall \overline{x}'.h(\overline{x}, \overline{x}') \equiv \neg\exists \overline{x}'.\neg h(\overline{x}, \overline{x}')$.

and so we can use `OBDDs` to represent and compute efficiently the switching functions in the algorithm above.

4.2 Abstraction Methods

When we study models with a strong hierarchical structure, we would want to exploit it to enhance the speed of the algorithms. To do so, we need to develop methods which preserve this structure.

But first of all, we need to formalize the notion of hierarchical structure. The example to keep in mind is that of a geographical map, divided in regions, provinces, cities and so on. This suggests the following definitions:

Definition 4.2.1 (Chain of partitions)**.** Consider a `KF` $\mathcal{X} = \langle X, \to, L\rangle$. We define a *chain of partitions* over $\mathcal{X}$ to be a finite sequence $\overline{\mathcal{P}} = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k)$ of partitions of X such that:

- $\mathcal{P}_0$ is the trivial partition $\{\{x\} | x \in X\}$.
- $\mathcal{P}_i \prec \mathcal{P}_{i+1}$ for each $i \in \{0, \ldots, k-1\}$
- for each partition $\mathcal{P}_i$, for each $S \in \mathcal{P}_i$ the `KF` $\mathcal{X}|_S = \langle S, \to|_{S\times S}, L|_{\mathtt{AP}\times S}\rangle$ is totally connected. Stated differently, for each pair $s, t \in S$ there exists an oriented path in S connecting s to t.

Remark 4.2.2. Note that given $S \in \mathcal{P}_j$, for $i < j$ the set $\{T \in \mathcal{P}_i | T \subseteq S\}$ is a partition of S.

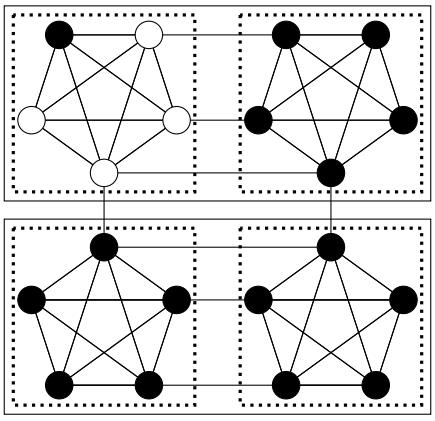
Definition 4.2.3. Consider a KF $\mathcal{X}$ and a chain of partitions $\overline{\mathcal{P}}$. Given $\mathcal{Q}$ a partition of X we say it is *compatible* with the chain if for all $S \in \mathcal{Q}$ there exists $\mathcal{P} \in \mathcal{P}$ such that $S \in \mathcal{P}$.

Moral 4.2.4. How can we interpret the objects introduced?

- A partition represents a subdivision of a spatial model in regions.
- A chain of partitions represents a hierarchical subdivision of the space model with several levels of *focus*. So a coarser partition corresponds to a subdivision in less but bigger regions, while a finer one corresponds to several small regions.

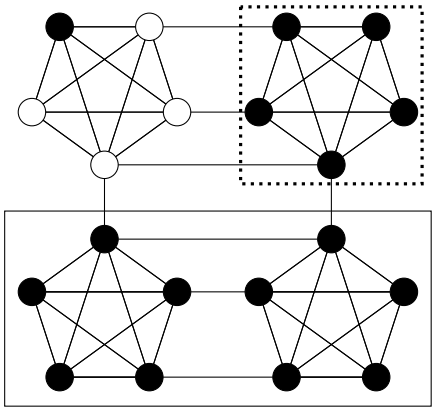
  Note that the compatibility request gives the hierarchical structure, as we can consider a region ($S \in \overline{\mathcal{P}}_{i+1}$) subdivided in smaller regions (the elements of $\mathcal{P}_i$ contained in S).
- A partition compatible with the chain is a mean to "compress" information: if we want to represent a set of states a way to do so is by using a subset of an appropriate compatible partition.

Example 4.2.5. Consider the KF represented in Figure 4.2.5, where with a dotted rectangles and continuous rectangles we indicated two partitions $\mathcal{P}_{\mathsf{dots}} \prec \mathcal{P}_{\mathsf{lines}}$.



If we want to "compress" the set of black nodes, we can do so by taking the compatible partition represented in Figure 4.2.5 and by taking an opportune subset of the partition (namely those subsets containing only black nodes).

In the rest of the chapter we will study how to "preserve" the hierarchical structure above while solving the model checking problem. The point is that knowing a priori a subdivision of the space gives us information on how to check a property. For example, if we want to go from Pisa to Rome by car we will first look for an highway from Tuscany to Lazio, then for a main road



near Pisa that connects it to the highway, and so on. Again, if we want to go from Pisa to Enna (in Sicily) by car we know we can't even before looking at the streets: it's common knowledge that there is no street connecting Sicily to the main land!

But how do we do this? We will try an approach similar to the abstraction-refinement heuristic presented in Chapter 2, so solving a model checking problem for an *approximated* model, thus obtaining information on how to *expand* the model to obtain a better approximation. So let's start by defining the approximation!

Definition 4.2.6 (Multi focus model)**.** A *multi focus model* is a tuple $\mathcal{X} = \langle X = C \sqcup A, \rightarrow, L \rangle$ where:

- X is a set of states divided in *concrete and abstract states* (states of C and A respectively).
- $\rightarrow$ is a reflexive adjacency relation.
- $L : X \rightarrow \{\top, ?, \bot\}$ is a 3-valued labeling function such that $L[C] \subseteq \{\top, \bot\}$ (while there is no restriction on the image of abstract states).

Moral 4.2.7. The idea behind `MFMs` is that, given a spatial model, we can hide some information about it, for example by packing groups of nodes together in a single abstract node. This makes us lose information on the structure of the sub-graph induced by the packed nodes, but at the same time allows to work with a smaller model.

Remark 4.2.8. The FN of a node (and of a set of nodes) is defined as in the `KF` case, with no distinction between concrete and abstract states. We can define as in the `KF` case the $\mathcal{C}$ and $\mathcal{B}^+$ operators.

Definition 4.2.9 (`MFM` subordinate to a partition)**.** Given a `KF` $\mathcal{X} = \langle X, \rightarrow, L \rangle$ and a partition $\mathcal{Q}$ of X we define the *`MFM` subordinate to* $\mathcal{Q}$ as $\mathcal{X}_\mathcal{Q} = \langle \mathcal{Q} = C \sqcup A, \rightarrow_\mathcal{Q}, L_\mathcal{Q} \rangle$ where

- $S \in C$ if and only if $|S| = 1$.
- $S \to_{\mathcal{Q}} T$ if and only if there exist states $s \in S$ and $t \in T$ such that $s \to t$.
- $L_{\mathcal{Q}}(p, S) = \begin{cases} \top & \text{if for all } s \in S \text{ it holds } L(p,s) = \top \\ \bot & \text{if for all } s \in S \text{ it holds } L(p,s) = \bot \\ ? & \text{otherwise} \end{cases}$

Remark 4.2.10. In agreement with the intuitive interpretation given above, we consider `KFs` as a special case of `MFMs` where all the states are concrete. Note that when we take the trivial partition we obtain again a `KF`.

In the rest of the document, we will consider only `MFM` subordinate to compatible partitions. What is the advantage? Doing so we have a mean to expand *locally* the model. More precisely:

Definition 4.2.11 (Local expansion)**.** Let $\mathcal{X}$ be a `KF`, $\overline{\mathcal{P}} = (\mathcal{P}_0, \dots)$ a chain of partitions and $\mathcal{Q}$ a partition compatible with the chain.

- Given $S \in \mathcal{Q}$ we define the *level* of S to be
$$\mathrm{lev}(S) = \min\{i | S \in \mathcal{P}_i\}$$
- Given $S \in \mathcal{Q}$ we define the *expansion of* $\mathcal{Q}$ *at* S to be the partition $\mathcal{Q}'$ obtained replacing S with its subsets belonging to level $\mathrm{lev}(S) - 1$. Formally:
$$\mathcal{Q}' = \begin{cases} \mathcal{Q} & \text{if } \mathrm{lev}(S) = 0 \\ (\mathcal{Q} \setminus \{S\}) \cup \{T \in \mathcal{P}_{\mathrm{lev}(S)-1} | T \subseteq S\} & \text{otherwise} \end{cases}$$
Note that $\mathcal{Q}'$ is again a partition compatible with the chain.

So using a chain of partitions we have a standard way to choose the approximations and to expand them gradually. Moreover, note that this choice is made so that the sub-graphs "hidden" by abstract nodes are totally connected, hence the following lemma:

Lemma 4.2.12 (Transfer)**.** *Let* $\mathcal{X}$ *and* $\mathcal{X}_{\mathcal{Q}}$ *as above. Consider a path* $\pi = (\pi_0, \dots, \pi_l)$ *of* $\mathcal{X}_{\mathcal{Q}}$*. Then for every pair* $s \in \pi_0, t \in \pi_l$ *there exists a path*
$$\sigma = (s = \sigma_0^{(0)}, \dots, \sigma_{k_0}^{(0)}; \sigma_0^{(1)}, \dots, \sigma_{k_1}^{(1)}; \dots; \sigma_0^{(l)}, \dots, \sigma_{k_l}^{(l)} = t)$$
of $\mathcal{X}$ *such that* $\sigma_j^{(i)} \in \pi_i$ *for every coherent choice of* i *and* j*.*

Now, following the abstraction-refinement method, we need:

- An *`MFM` semantics* for `SLCS` extending the one already introduced.
- An *algorithm* to compute the semantics.
- A way to *retrieve information* on which states to expand.

4.2.1 The Semantics

In this subsection we want to define the MFM semantics for SLCS. More specifically, for each MFM $\mathcal{M} = \langle S = C \sqcup A, \rightarrow, L\rangle$ we want to define a relation $[\models] \subseteq S \times \{\top, ?, \bot\} \times \mathcal{F}_{\texttt{SLCS}}$ (where $\mathcal{F}_{\texttt{SLCS}}$ is the set of the SLCS formulas) such that

$$s \models_\theta \varphi \quad \text{for } s \in S, \theta \in \{\top, ?, \bot\}, \varphi \in \mathcal{F}_{\texttt{SLCS}}$$

means informally that the state s thinks that the formula φ has value θ.

We introduce here some notations to improve the readability of the next definitions.

Notation 4.2.13.

- We define a *state condition* as a pair $\langle \varphi, \theta\rangle$ where φ is a SLCS formula and $\Theta \subseteq \{\top, ?, \bot\}$. We say a state s *respects* the condition above if $s \models_\theta \varphi$ for some $\theta \in \Theta$. Of course, this notation depends on the semantics $[\models]$ not yet introduced.

  Usually for Θ we will use a simplified notation (i.e., we will write $\langle \varphi, \theta\rangle$ for the condition $\langle \varphi, \{\theta\}\rangle$ and $\langle \varphi, \geq \theta\rangle$ for the condition $\langle \varphi, \{\theta' | \theta' \geq \theta\}\rangle$).

- Given two conditions Φ and Ψ, we define a path π of an MFM to be a (Φ, Ψ)-*path* if the followings hold:

  1. $l = \text{length}(\pi) > 0$.
  2. For all $k < l$, π_k respects Φ.
  3. π_l respects Ψ.

- We indicate with $\Pi_s(\Phi, \Psi)$ the set of (Φ, Ψ)-paths starting at s.

- We call a path π a $(\Phi, \Psi)^+$-path if the properties above hold, but restricting property 2 to the values $0 < k < l$. We indicate with $\Pi_s^+(\Phi, \Psi)$ the set of $(\Phi, \Psi)^+$-paths starting at s.

  Note that this this definition strongly resembles Definition 3.1.23.

Definition 4.2.14 (MFM semantics for SLCS)**.** Given an MFM $\mathcal{X} = \langle X = C \sqcup A, \rightarrow, L\rangle$, given a state $s \in X$, given a formula φ, we want to define the entailment relation $\mathcal{X}, s \models_\theta \varphi$ (meaning φ has value θ at state s), so that a formula at a state assume only one value. In the following we will denote

- with $[\![\varphi]\!]_\theta^{\mathcal{X}}$ the set of states at whom the formula has value θ.
- with $[\![\varphi]\!]_{\leq\theta}^{\mathcal{X}}$ the set $\bigcup_{\theta' \leq \theta} [\![\varphi]\!]_{\theta'}^{\mathcal{X}}$ (and a similar definition for $[\![\varphi]\!]_{\geq\theta}^{\mathcal{X}}$).
- with $[\![\varphi, s]\!]$ the value of φ at s.

We define the entailment by induction over the structure of the formula φ. We divide the inductive definition in two parts, boolean connectives (and cases) and spatial operators.

Boolean Connectives

$$
\begin{array}{l}
\mathcal{X}, s \nvDash_{\top} \bot \\
\mathcal{X}, s \nvDash_{?} \bot \\
\mathcal{X}, s \vDash_{\bot} \bot
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{X}, s \vDash_{\top} p \in \mathtt{AP} & \iff & L(p,s) = \top \\
\mathcal{X}, s \vDash_{?} p \in \mathtt{AP} & \iff & L(p,s) = ? \\
\mathcal{X}, s \vDash_{\bot} p \in \mathtt{AP} & \iff & L(p,s) = \bot
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{X}, s \vDash_{\top} \psi_1 \wedge \psi_2 & \iff & [\![\psi_1, s]\!] \wedge [\![\psi_2, s]\!] = \top \\
\mathcal{X}, s \vDash_{?} \psi_1 \wedge \psi_2 & \iff & [\![\psi_1, s]\!] \wedge [\![\psi_2, s]\!] = ? \\
\mathcal{X}, s \vDash_{\bot} \psi_1 \wedge \psi_2 & \iff & [\![\psi_1, s]\!] \wedge [\![\psi_2, s]\!] = \bot
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{X}, s \vDash_{\top} \neg\psi & \iff & \mathcal{X}, s \vDash_{\bot} \psi \\
\mathcal{X}, s \vDash_{?} \neg\psi & \iff & \mathcal{X}, s \vDash_{?} \psi \\
\mathcal{X}, s \vDash_{\bot} \neg\psi & \iff & \mathcal{X}, s \vDash_{\top} \psi
\end{array}
$$

Spatial Operators We will separate the cases when $s = c \in C$ and $s = a \in A$ for the $\mathcal{N}$ operator.

$$
\begin{array}{lcl}
\mathcal{X}, c \vDash_{\top} \mathcal{N}(\psi) & \iff & c \in \mathrm{FN}\left([\![\psi]\!]_{\top}\right) \\
\mathcal{X}, c \vDash_{?} \mathcal{N}(\psi) & \iff & c \in \mathrm{FN}\left([\![\psi]\!]_{?}\right) \setminus \mathrm{FN}\left([\![\psi]\!]_{\top}\right) \\
\mathcal{X}, c \vDash_{\bot} \mathcal{N}(\psi) & \iff & c \notin \mathrm{FN}\left([\![\psi]\!]_{\geq ?}\right)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{X}, a \vDash_{\top} \mathcal{N}(\psi) & \iff & a \in [\![\psi]\!]_{\top} \\
\mathcal{X}, a \vDash_{?} \mathcal{N}(\psi) & \iff & a \in \mathrm{FN}\left([\![\psi]\!]_{\geq ?}\right) \setminus [\![\psi]\!]_{\top} \\
\mathcal{X}, a \vDash_{\bot} \mathcal{N}(\psi) & \iff & a \notin \mathrm{FN}\left([\![\psi]\!]_{\geq ?}\right)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{X}, c \vDash_{\top} \mathcal{S}(\psi_1, \psi_2) & \iff & \text{there exists } S \subseteq X \text{ such that } c \in S \text{ and} \\
 & & \left\{ \begin{array}{rcl} S & \subseteq & [\![\psi_1]\!]_{\top} \\ \mathcal{B}^{+}(S) & \subseteq & [\![\psi_2]\!]_{\top} \end{array} \right. \\
\mathcal{X}, c \vDash_{\bot} \mathcal{S}(\psi_1, \psi_2) & \iff & c \vDash_{\bot} \psi_1 \text{ or } \Pi_c^{+}\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right) \neq \emptyset \\
\mathcal{X}, c \vDash_{?} \mathcal{S}(\psi_1, \psi_2) & \iff & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{X}, a \vDash_{\top} \mathcal{S}(\psi_1, \psi_2) & \iff & \text{there exists } S \subseteq X \text{ such that } a \in S \text{ and} \\
 & & \left\{ \begin{array}{rcl} S & \subseteq & [\![\psi_1]\!]_{\top} \\ \mathcal{B}^{+}(S) & \subseteq & [\![\psi_2]\!]_{\top} \end{array} \right. \\
\mathcal{X}, a \vDash_{\bot} \mathcal{S}(\psi_1, \psi_2) & \iff & a \vDash_{\bot} \psi_1 \text{ or } \Pi_a\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right) \neq \emptyset \\
\mathcal{X}, a \vDash_{?} \mathcal{S}(\psi_1, \psi_2) & \iff & \text{otherwise}
\end{array}
$$

Remark 4.2.15. Some cases in the definition above are not well-defined because they rely on the definition of entailment for the same formula (for example $s \vDash_{?} \mathcal{S}(\psi_1, \psi_2)$ depends on the definition of $s \vDash_{\top} \mathcal{S}(\psi_1, \psi_2)$ and

$s \vDash_\bot \mathcal{S}(\psi_1, \psi_2)$). To solve the problem, the induction has to be performed over the set $\{\langle \varphi, \theta \rangle \,|\varphi \in \texttt{SLCS}, \theta \in \{\top, ?, \bot\}\}$ using a slightly different well-founded relation $\preceq$, namely

$$\langle \varphi, \theta \rangle \preceq \langle \psi, \theta' \rangle \iff \varphi \text{ is less complex than } \psi \text{ or } [\varphi \equiv \psi] \wedge \begin{cases} \theta \geq' \theta' & \text{if } \varphi \equiv \mathcal{S}(\varphi_1, \varphi_2) \\ \theta \geq \theta' & \text{otherwise} \end{cases}$$

where $\bot \leq ? \leq \top$ and $? \leq' \bot \leq' \top$.

Moral 4.2.16. The idea behind this semantics is that an `MFM` represents a `KF` where some strongly connected sub-graphs have been identified in a single node (abstract states), while other nodes of the `KF` are left untouched (concrete states). So, given $\mathcal{N}$ an `MFM`, if we want to prove a transfer property similar to 2.3.11, the semantics can give value $\top$ or $\bot$ to a state s of $\mathcal{N}$ if and only if for each `KF` $\mathcal{M}$ that can be represented by $\mathcal{N}$, the property holds for all the states represented by s.

With this reasoning in mind we can explain the semantics given to spatial operators:

- **Case** $s \vDash_\top \mathcal{N}(\psi)$**:** proving that s entails the formula $\mathcal{N}(\psi)$ means that *each* state represented by s entails the formula. So proving $c \vDash_\top \mathcal{N}(\psi)$ for a concrete state c simply means to show it is in $\mathcal{C}(\llbracket \psi \rrbracket_\top)$.

  For an abstract state a the situation is quite different, since proving $a \in \mathcal{C}(\llbracket \psi \rrbracket_\top)$ simply means that *one of* the states represented by a entails $\mathcal{N}(\psi)$. In this case, the only way to be sure that *all* the states represented entail $\mathcal{N}(\psi)$, is to prove that a itself proves ψ, since a node is always contained in its own closure.

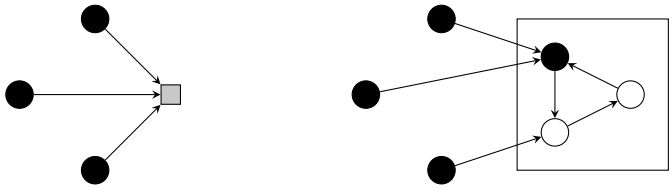


Figure 4.1: On the left, a portion of an `MFM`. The colors of the nodes indicate the value of the formula ψ at each node (black means $\top$, gray means ? and white means $\bot$). On the right, a possible `KF` it represents. Note that the abstract state *can't entail* $\mathcal{N}(\psi)$ as in the instantiation there is a node which doesn't entail the formula.

- **Case** $s \vDash_\bot \mathcal{N}(\psi)$**:** this case is quite simple, since to prove that each state represented by s doesn't entail $\mathcal{N}(\psi)$ we simply have to make sure $s \notin \llbracket \psi \rrbracket_{\geq ?}$.

- **Case** $s \vDash_\top \mathcal{S}(\psi_1, \psi_2)$**:** to prove that each node represented by s is in a *safe zone* (meaning in a set S as in Definition 3.1.9) we can prove that s itself is in a safe zone S'. The only thing to verify is that the *border* of a safe zone S' in the `MFM` contains the border of the corresponding safe zone S in the `KF` it represents.

- **Case** $s \vDash_\bot \mathcal{S}(\psi_1, \psi_2)$**:** this is the trickiest case and the main reason we ask the sub-graphs represented by abstract nodes to be strongly connected. In a `KF` we already proved that the condition $t \nvDash \mathcal{S}(p, q)$ corresponds to the existence of a path with certain properties (see Lemma 3.1.24).

  Now, if we find a path π in the `MFM`, by the strongly connectivity this corresponds to a path π' in the `KF` represented, and so the conditions in the semantics above are made to ensure π' has the properties needed to disprove $\mathcal{S}(\psi_1, \psi_2)$.

Now, let us prove that the semantics is well defined.

Lemma 4.2.17. *The semantics above is well-defined, i.e. for every formula φ and every state s it holds* one and only one *of the following:*

$$s \vDash_\top \varphi$$
$$s \vDash_? \varphi$$
$$s \vDash_\bot \varphi$$

Proof. We prove the result by induction on the structure of the formula. Note that the boolean cases are trivial, so we prove the result only for the spatial operators:

- **Case** $\varphi \equiv \mathcal{N}(\psi)$**:** Is trivial to show that $[\![\mathcal{N}(\psi)]\!]_\top$ and $[\![\mathcal{N}(\psi)]\!]_?$ are disjoint. The result follows then by the following inclusions.

$$[\![\psi]\!]_\top \subseteq \mathrm{FN}\left([\![\psi]\!]_\top\right) \subseteq \mathrm{FN}\left([\![\psi]\!]_{\geq ?}\right)$$

- **Case** $\varphi \equiv \mathcal{S}(\psi_1, \psi_2)$**:** First of all, note that $\Pi_s(\Phi, \Psi) \subseteq \Pi_s^+(\Phi, \Psi)$, so we have to show that the conditions

  1. There exists S such that $s \in S$ and $\begin{cases} S & \subseteq & [\![\psi_1]\!]_\top \\ \mathcal{B}^+(S) & \subseteq & [\![\psi_2]\!]_\top \end{cases}$
  2. $s \vDash_\bot \psi_1$ or $\Pi_s^+\left(\langle \psi_2, \bot\rangle, \langle \psi_1 \vee \psi_2, \bot\rangle\right) \neq \emptyset$

  are incompatible. But with the same proof of Lemma 3.1.24 we have that condition 1 is equivalent to

$$s \vDash_\top \psi_1 \text{ and } \Pi_s^+\left(\langle \psi_2, \leq ?\rangle, \langle \psi_1 \vee \psi_2, \leq ?\rangle\right) = \emptyset$$

  and now is immediate to verify that conditions 1 and 2 are incompatible.

□

Now, let's prove that this semantics extends the KF one.

Lemma 4.2.18. *Consider an MFM with no abstract states $\mathcal{M} = \langle X, \to, L\rangle$ and an SLCS formula φ. Then*

$$\llbracket\varphi\rrbracket^{KF} = \llbracket\varphi\rrbracket^{MFM}_{\top}$$
$$\overline{\llbracket\varphi\rrbracket^{KF}} = \llbracket\varphi\rrbracket^{MFM}_{\bot}$$

where with $\llbracket\bullet\rrbracket^{KF}$ we indicate the usual KF semantics and with $\llbracket\bullet\rrbracket^{MFM}$ we indicate the semantics introduced here.

Proof. By induction over the formula φ:

- **Case $\varphi \equiv p \in \mathtt{AP}$:** the result follows trivially as

$$\llbracket\varphi\rrbracket^{\mathtt{MFM}}_{\top} = L_p^{-1}(\top) = \llbracket\varphi\rrbracket^{\mathtt{KF}}$$
$$\llbracket\varphi\rrbracket^{\mathtt{MFM}}_{\bot} = L_p^{-1}(\bot) = \overline{\llbracket\varphi\rrbracket^{\mathtt{KF}}}$$

  Note that here we used the property of MFMs $L[C] \subseteq \{\top, \bot\}$.

- **Cases $\varphi \equiv \bot, \psi_1 \wedge \psi_2, \neg\psi$:** trivial.

- **Case $\varphi \equiv \mathcal{N}(\psi)$:** by inductive hypothesis we have:

$$\llbracket\psi\rrbracket^{\mathtt{MFM}}_{\top} = \llbracket\psi\rrbracket^{\mathtt{KF}}$$
$$\llbracket\psi\rrbracket^{\mathtt{MFM}}_{\bot} = \overline{\llbracket\psi\rrbracket^{\mathtt{KF}}}$$
$$\llbracket\psi\rrbracket^{\mathtt{MFM}}_{?} = \emptyset$$

  And so it follows:

$$\llbracket\mathcal{N}(\psi)\rrbracket^{\mathtt{MFM}}_{\top} = \mathrm{FN}(\llbracket\psi\rrbracket^{\mathtt{MFM}}_{\top}) = \mathrm{FN}(\llbracket\psi\rrbracket^{\mathtt{KF}}) = \llbracket\mathcal{N}(\psi)\rrbracket^{\mathtt{KF}}$$
$$\llbracket\mathcal{N}(\psi)\rrbracket^{\mathtt{MFM}}_{?} = \mathrm{FN}(\llbracket\psi\rrbracket_{?}) \setminus \mathrm{FN}(\llbracket\psi\rrbracket_{\top}) \subseteq \mathrm{FN}(\emptyset) = \emptyset$$
$$\llbracket\mathcal{N}(\psi)\rrbracket^{\mathtt{MFM}}_{\bot} = \overline{\llbracket\mathcal{N}(\psi)\rrbracket^{\mathtt{MFM}}_{\top} \cup \llbracket\mathcal{N}(\psi)\rrbracket^{\mathtt{MFM}}_{?}} = \overline{\llbracket\mathcal{N}(\psi)\rrbracket^{\mathtt{KF}}}$$

- **Case $\varphi \equiv \mathcal{S}(\psi_1, \psi_2)$:** by inductive hypothesis:

$$\llbracket\psi_i\rrbracket^{\mathtt{MFM}}_{\top} = \llbracket\psi_i\rrbracket^{\mathtt{KF}}$$
$$\llbracket\psi_i\rrbracket^{\mathtt{MFM}}_{\bot} = \overline{\llbracket\psi_i\rrbracket^{\mathtt{KF}}}$$
$$\llbracket\psi_i\rrbracket^{\mathtt{MFM}}_{?} = \emptyset$$

for $i \in \{1, 2\}$. Then for s a state, by Lemma 3.1.24 it follows:

$$\begin{aligned} s \models_{\top}^{\texttt{MFM}} \mathcal{S}(\psi_1, \psi_2) \iff & \text{ there exists } S \subseteq X \text{ such that } s \in S \text{ and} \\ & \left\{ \begin{array}{rcl} S & \subseteq & [\![\psi_1]\!]_{\top}^{\texttt{MFM}} = [\![\psi_1]\!]^{\texttt{KF}} \\ \mathcal{B}^+(S) & \subseteq & [\![\psi_2]\!]_{\top}^{\texttt{MFM}} = [\![\psi_2]\!]^{\texttt{KF}} \end{array} \right. \\ \iff & s \models^{\texttt{KF}} \mathcal{S}(\psi_1, \psi_2) \end{aligned}$$

$$\begin{aligned} s \models_{\bot}^{\texttt{MFM}} \mathcal{S}(\psi_1, \psi_2) &\iff s \models_{\bot}^{\texttt{MFM}} \psi_1 \text{ or } \Pi_s^+ \left(\langle \psi_2, \bot \rangle \,.\, \langle \psi_1 \vee \psi_2, \bot \rangle \right) \neq \emptyset \\ &\iff s \models^{\texttt{KF}} \neg\psi_1 \text{ or } \Pi_s^+ \left(\neg\psi_2, \neg\psi_1 \wedge \neg\psi_2 \right) \neq \emptyset \\ &\iff s \not\models^{\texttt{KF}} \mathcal{S}(\psi_1, \psi_2) \end{aligned}$$

where of course $s \models_{\theta}^{\texttt{MFM}} \chi$ means $s \in [\![\chi]\!]_{\theta}^{\texttt{MFM}}$ and $s \models^{\texttt{KF}} \chi$ means $s \in [\![\chi]\!]^{\texttt{KF}}$.

□

Now that we showed the semantics introduced is a proper extension of the KF semantics, we have to show *why* we introduced it. In particular we want to prove a *transfer property* similar to the one introduced for CTL.

Theorem 4.2.19 (Transfer property)**.** *Let $\mathcal{M} = \langle X, \to, L \rangle$ be a KF and $\mathcal{Q} \prec \mathcal{Q}'$ two partitions of X such that for each $S \in \mathcal{Q} \cup \mathcal{Q}'$ then $\langle S, \to|_{S \times S} \rangle$ is strongly connected. Let $\mathcal{M}_{\mathcal{Q}} = \langle \mathcal{Q} = C \sqcup A, \to_{\mathcal{Q}}, L_{\mathcal{Q}} \rangle$ and $\mathcal{M}_{\mathcal{Q}'} = \langle \mathcal{Q}' = C' \sqcup A', \to_{\mathcal{Q}'}, L_{\mathcal{Q}'} \rangle$ the MFMs corresponding to $\mathcal{Q}$ and $\mathcal{Q}'$ respectively. Let $h : \mathcal{M}_{\mathcal{Q}} \to \mathcal{M}_{\mathcal{Q}'}$ such that*

$$h\left([s]_{\mathcal{Q}}\right) = [s]_{\mathcal{Q}'}$$

Then for $S' \in \mathcal{Q}'$, for $\varphi \in$ SLCS it holds:

$$\begin{array}{lll} S' \models_{\top} \varphi & \implies & \text{for all } S \in h^{-1}(S')\text{: } S \models_{\top} \varphi \\ S' \models_{\bot} \varphi & \implies & \text{for all } S \in h^{-1}(S')\text{: } S \models_{\bot} \varphi \end{array}$$

In particular, if $\mathcal{Q}$ is the trivial partition, we obtain:

$$\begin{array}{lll} S' \models_{\top} \varphi & \implies & \text{for all } s \in S'\text{: } s \models_{\top} \varphi \\ S' \models_{\bot} \varphi & \implies & \text{for all } s \in S'\text{: } s \models_{\bot} \varphi \end{array}$$

Remark 4.2.20. Note that h is well defined because $\mathcal{Q} \prec \mathcal{Q}'$.

The proof of this Theorem relies on some technical results that it's useful to state separately.

Lemma 4.2.21. *Let $\mathcal{M}$, $\mathcal{Q}$, $\mathcal{Q}'$, $\mathcal{M}_{\mathcal{Q}}$, $\mathcal{M}_{\mathcal{Q}'}$ and h as above. Then:*

1. *for* $S \in \mathcal{Q}$ *and* $\mathfrak{T} \subseteq \mathcal{Q}$

$$S \notin \mathrm{FN}(\mathfrak{T}) \iff \textit{for all } s \in S,\ s \notin \mathrm{FN}\left(\bigcup \mathfrak{T}\right)$$

and so

$$S \in \mathrm{FN}(\mathfrak{T}) \iff \textit{there exists } s \in S \textit{ such that } s \in \mathrm{FN}\left(\bigcup \mathfrak{T}\right)$$

2. *for* $\mathfrak{T} \subseteq \mathcal{Q}'$

$$\mathrm{FN}\left(h^{-1}(\mathfrak{T})\right) \subseteq h^{-1}\left(\mathrm{FN}(\mathfrak{T})\right)$$

3. *Given a path* π' *of* $\mathcal{M}_{\mathcal{Q}'}$ *and a state* S *of* $\mathcal{M}_{\mathcal{Q}}$ *such that* $h(S) = \pi'_0$, *there exists a path* π *of* $\mathcal{M}_{\mathcal{Q}}$ *starting at* S *such that:*

$$\begin{cases} \pi' = (\pi'_0, \ldots, \pi'_l) \\ \pi = \left(S = \pi^0_0, \ldots, \pi^0_{k_0}; \pi^1_0, \ldots, \pi^1_{k_1}; \ldots; \pi^l_0, \ldots, \pi^l_{k_l}\right) \\ \textit{for each } i \in \{0, \ldots, l\},\ \textit{for each } j \in \{0, \ldots, k_i\} \textit{ it holds } h(\pi^i_j) = \pi'_i \end{cases}$$

Moreover, if $\pi'_j \in C'$, *then we can choose* π *so that* $k_j = 0$ *(meaning the* j*th portion of the path* π *contains only one node).*

Notation 4.2.22. For π and π' as in the point 3, we say that π' can be *lifted* to π.

Proof.

1. It follows directly from the definition of abstraction:

$$\begin{aligned} S \notin \mathrm{FN}(\mathfrak{T}) &\iff \text{for all } T \in \mathfrak{T},\ T \not\to_{\mathcal{Q}} S \\ &\iff \text{for all } T \in \mathfrak{T}, \text{for all } s \in S, \text{for all } t \in T,\ t \not\to s \\ &\iff \text{for all } s \in S, \text{for all } t \in \bigcup \mathfrak{T},\ t \not\to s \\ &\iff \text{for all } s \in S, s \notin \mathrm{FN}\left(\bigcup \mathfrak{T}\right) \end{aligned}$$

2. This follows trivially from the definition of `MFM` subordinate to a partition:

$$\begin{aligned} S \in \mathrm{FN}(h^{-1}(\mathfrak{T})) &\iff \text{there exists } T \in h^{-1}(\mathfrak{T}) \text{ such that } T \to_{\mathcal{Q}} S \\ &\iff \text{there exists } t \in \bigcup h^{-1}(\mathfrak{T}),\ s \in S \text{ such that } t \to s \\ &\iff \text{there exists } t \in \bigcup \mathfrak{T},\ s \in S \text{ such that } t \to s \\ &\implies \text{there exists } t \in \bigcup \mathfrak{T},\ s \in h(S) \text{ such that } t \to s \\ &\iff h(S) \in \mathrm{FN}(\mathfrak{T}) \\ &\iff s \in h^{-1}\left(\mathrm{FN}(\mathfrak{T})\right) \end{aligned}$$

3. Direct consequence of the strong connection of the elements in the partitions $\mathcal{Q}$ and $\mathcal{Q}'$. The last assertion follows by a simple cutting-and-gluing argument.

□

Proof of Lemma 4.2.19. Fix S' and $S \in h^{-1}(S')$. We prove the result by induction over the structure of the formula. Of course, the cases for boolean operators are trivial, so for brevity we prove the result only for spatial operators.

- **Case $\varphi \equiv \mathcal{N}(\psi)$ and $S' \vDash_\top \varphi$:** by definition of the semantics we have two cases to consider, if $S' \in C$ or if $S' \in A$.
  - **If** $S' \in C$, then by the definition of $\mathcal{M}_{\mathcal{Q}'}$ we have that
    $$S' \in C' \iff \left|S'\right| = 1 \implies |S| = 1 \iff S \in C$$
    and so in this case both states are concrete ones. Let S' and S be the singleton $\{s\}$. By inductive hypothesis and Lemma 4.2.21 it follows:
    $$\begin{aligned} S' \vDash_\top \mathcal{N}(\psi) &\iff S' \in \mathrm{FN}\left([\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}'}}\right) \\ &\iff \text{there exists } T' \in [\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}'}} \text{ such that } T' \to_{\mathcal{Q}'} S' \\ &\iff \text{there exists } t \in \bigcup [\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}'}} \text{ such that } t \to s \\ &\implies \text{there exists } t \in \bigcup [\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}}} \text{ such that } t \to s \\ &\iff \text{there exists } T \in [\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}}} \text{ such that } T \to_{\mathcal{Q}} S \\ &\iff S \in \mathrm{FN}\left([\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}}}\right) \\ &\iff S \vDash_\top \mathcal{N}(\psi) \end{aligned}$$
  - **If** $S' \in A$ then by inductive hypothesis
    $$\begin{aligned} S' \vDash_\top \mathcal{N}(\psi) &\iff S' \in [\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}'}} \\ &\implies S \in [\![\psi]\!]_\top^{\mathcal{M}_{\mathcal{Q}}} \\ &\implies S \vDash_\top \mathcal{N}(\psi) \end{aligned}$$
- **Case $\varphi \equiv \mathcal{N}(\psi)$ and $S' \vDash_\bot \varphi$:** By Lemma 4.2.21 and inductive hypothesis we have:
  $$\begin{aligned} S' \vDash_\bot \mathcal{N}(\psi) &\iff S' \notin \mathrm{FN}\left([\![\psi]\!]_{\geq ?}^{\mathcal{M}_{\mathcal{Q}'}}\right) = \mathrm{FN}\left(\overline{[\![\psi]\!]_\bot^{\mathcal{M}_{\mathcal{Q}'}}}\right) \\ &\iff \text{for all } s \in S',\ s \notin \mathrm{FN}\left(\bigcup \overline{[\![\psi]\!]_\bot^{\mathcal{M}_{\mathcal{Q}'}}}\right) \\ &\implies \text{for all } s \in S,\ s \notin \mathrm{FN}\left(\bigcup \overline{[\![\psi]\!]_\bot^{\mathcal{M}_{\mathcal{Q}}}}\right) \\ &\iff S \notin \mathrm{FN}\left(\overline{[\![\psi]\!]_\bot^{\mathcal{M}_{\mathcal{Q}}}}\right) = \mathrm{FN}\left([\![\psi]\!]_{\geq ?}^{\mathcal{M}_{\mathcal{Q}}}\right) \\ &\iff S \vDash_\bot \mathcal{N}(\psi) \end{aligned}$$

- **Case $\varphi \equiv \mathcal{S}(\psi_1, \psi_2)$ and $S' \vDash_\top \varphi$:** By Lemma 4.2.21 and inductive hypothesis we have:

$$
\begin{array}{rcl}
S' \vDash_\top \mathcal{S}(\psi_1, \psi_2) & \iff & \text{there exists } \mathfrak{T}' \subseteq \mathcal{Q}' \text{ such that } S' \in \mathfrak{T}' \text{ and} \\
& & \begin{cases} \mathfrak{T}' & \subseteq & [\![\psi_1]\!]_\top^{\mathcal{M}_{\mathcal{Q}'}} \\ \mathcal{B}^+(\mathfrak{T}') & \subseteq & [\![\psi_2]\!]_\top^{\mathcal{M}_{\mathcal{Q}'}} \end{cases} \\
& \implies & \text{there exists } \mathfrak{T}' \subseteq \mathcal{Q}' \text{ such that } S \in h^{-1}(\mathfrak{T}') \\
& & \text{and } \begin{cases} h^{-1}(\mathfrak{T}') & \subseteq & [\![\psi_1]\!]_\top^{\mathcal{M}_{\mathcal{Q}}} \\ h^{-1}\left(\mathcal{B}^+(\mathfrak{T}')\right) & \subseteq & [\![\psi_2]\!]_\top^{\mathcal{M}_{\mathcal{Q}}} \end{cases} \\
& \implies & \text{there exists } \mathfrak{T} \subseteq \mathcal{Q} \text{ such that } S \in \mathfrak{T} \text{ and} \\
& & \begin{cases} \mathfrak{T} & \subseteq & [\![\psi_1]\!]_\top^{\mathcal{M}_{\mathcal{Q}}} \\ \mathcal{B}^+(\mathfrak{T}) & \subseteq & [\![\psi_2]\!]_\top^{\mathcal{M}_{\mathcal{Q}}} \end{cases} \\
& \iff & S \vDash_\top \mathcal{S}(\psi_1, \psi_2)
\end{array}
$$

- **Case $\varphi \equiv \mathcal{S}(\psi_1, \psi_2)$ and $S' \vDash_\bot \varphi$:** by definition of the semantics we have two cases to consider, if $S' \in C'$ or if $S' \in A'$.
  - **If $S' \in C'$**, then by definition we have

    $$S' \vDash_\bot \mathcal{S}(\psi_1, \psi_2) \iff S' \vDash_\bot \psi_2 \text{ or } \Pi_{S'}^+\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right) \neq \emptyset$$

    If the first disjunct holds, then by inductive hypothesis it holds also for S.
    Otherwise, let $\pi' \in \Pi_{S'}^+\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right)$. By Lemma 4.2.21 we can find a path π of $\mathcal{M}_{\mathcal{Q}}$ starting at S which lifts to π', and it is not restrictive to suppose $h(\pi_l) \neq h(\pi_{l-1})$ for $l = \text{length}(\pi)$. Note also that as $S' \in C'$ then (again by Lemma 4.2.21) we can choose π so that $h(\pi_1) = \pi'_1$.
    By inductive hypothesis it follows trivially that
    $\pi \in \Pi_S^+\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right)$ and so the thesis.
  - **If $S' \in A'$**, then the reasoning above applies with little modifications. In particular, given $\pi' \in \Pi_{S'}\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right)$ then there exists

    $$\pi \in \Pi_S\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right) \subseteq \Pi_S^+\left(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle\right)$$

    and so the thesis follows both in the case $S \in C$ and $S \in A$.

□

Remark 4.2.23. This result is really important to develop an abstraction-refinement approach. In fact now we know that by solving the `MFM` model checking problem for $\mathcal{M}_{\mathcal{Q}}$ we are gathering information on the solution of the `KF` model checking problem for $\mathcal{M}$.

Moreover, now we can refine the model $\mathcal{M}_{\mathcal{Q}}$ step-by-step by choosing a subset of $\mathcal{Q}$ and by expanding locally the model (as in Definition 4.2.11). The theorem above tells us that we are not losing information by doing so.

4.2.2 The Algorithm

In this Subsection we fix an MFM $\mathcal{M} = \langle X = C \sqcup A, \rightarrow, L \rangle$.

We now present an algorithm to compute the semantics for SLCS fixed the MFM model. As usual, we give only explicit algorithms to compute the semantics of formulas $\mathcal{N}(p)$ and $\mathcal{S}(p,q)$ for $p, q \in \mathtt{AP}$, since a complete algorithm follows trivially.

Algorithm 14: Algorithm to compute $\mathcal{N}(p)$

input : $\mathcal{M}$, $\mathcal{N}(p)$
output: $[\![\mathcal{N}(p)]\!]_\top$, $[\![\mathcal{N}(p)]\!]_?$, $[\![\mathcal{N}(p)]\!]_\bot$

1 nearTrue $:= (C \cap \mathrm{FN}([\![p]\!]_\top)) \cup (A \cap [\![p]\!]_\top)$;
2 nearMaybe $:= \mathrm{FN}\left([\![p]\!]_{\geq ?}\right) \setminus$ nearTrue ;
3 nearFalse $:= X \setminus \mathrm{FN}\left([\![p]\!]_{\geq ?}\right)$;
4 **return** nearTrue, nearMaybe, nearFalse

Algorithm 15: Algorithm to compute $\mathcal{S}(p,q)$

input : $\mathcal{M}$, $\mathcal{S}(p,q)$
output: $[\![\mathcal{S}(p,q)]\!]_\top$, $[\![\mathcal{S}(p,q)]\!]_?$, $[\![\mathcal{S}(p,q)]\!]_\bot$

1 surrTrue $:= [\![p]\!]_\top$;
2 surrMaybe $:= [\![p]\!]_?$;
3 surrFalse $:= [\![p]\!]_\bot$;
4 badBorder $:= [\![p]\!]_{\leq ?} \cap [\![q]\!]_{\leq ?}$;
5 **repeat**
6 corrosion $:=$ surrTrue $\cap \mathrm{BN}($badBorder$)$;
7 surrTrue $:=$ surrTrue $\setminus$ corrosion ;
8 surrMaybe $:=$ surrMaybe $\cup$ corrosion ;
9 badBorder $:= [\![q]\!]_{\leq ?} \cap$ corrosion ;
10 **until** badBorder $== \emptyset$;
11 badBorder $:= [\![p]\!]_\bot \cap [\![q]\!]_\bot$;
12 badCandidates $:= C \cup (A \cap [\![q]\!]_\bot) \cap$ surrMaybe ;
13 **repeat**
14 corrosion $:=$ badCandidates $\cap \mathrm{BN}($badBorder$)$;
15 badCandidates $:=$ badCandidates $\setminus$ corrosion ;
16 surrMaybe $:=$ surrMaybe $\setminus$ corrosion ;
17 surrFalse $:=$ surrFalse $\cup$ corrosion ;
18 badBorder $:=$ corrosion $\cap [\![q]\!]_\bot$;
19 **until** badBorder $== \emptyset$;
20 **return** surrTrue, surrMaybe, surrFalse

Theorem 4.2.24. *The Algorithms 14 and 15 terminate correctly and have linear-time complexity on the encoding of* $\mathcal{M}$.

Proof. Let us examine one algorithm at a time:

Case $\mathcal{N}(p)$: Let us prove termination, correctness and computational time separately.

Termination: The algorithm terminates as there is no recursion.

Correctness: This follows trivially from the definition of the semantics. If $c \in C$ and $a \in A$:

$$
\begin{aligned}
c \in \mathsf{nearTrue} &\iff c \in \mathrm{FN}\left([\![p]\!]_{\top}\right) \\
c \in \mathsf{nearMaybe} &\iff c \in \mathrm{FN}\left([\![p]\!]_{\geq ?}\right) \setminus \mathrm{FN}\left([\![p]\!]_{\top}\right) \\
c \in \mathsf{nearFalse} &\iff c \notin \mathrm{FN}\left([\![p]\!]_{\geq ?}\right) \\
a \in \mathsf{nearTrue} &\iff a \in [\![p]\!]_{\top} \\
a \in \mathsf{nearMaybe} &\iff a \in \mathrm{FN}\left([\![p]\!]_{\geq ?}\right) \setminus [\![p]\!]_{\top} \\
a \in \mathsf{nearFalse} &\iff a \notin \mathrm{FN}\left([\![p]\!]_{\geq ?}\right)
\end{aligned}
$$

in accord with the semantics of $\mathcal{N}$.

Complexity: the algorithm has linear complexity because all operations used in the pseudo-code (union, intersection, set difference, forward neighborhood) have linear-time complexity.

Case $\mathcal{S}(p, q)$: Let us prove termination, correctness and computational time separately.

Termination: There are two recursions in the algorithm, so we have to prove that both terminate.

For the first recursion, we have the followings facts:

- During a cycle, if $\mathsf{corrosion} = \emptyset$ (line 6) then $\mathsf{badBorder} = \emptyset$ (line 9) and so the recursion terminates.
- Otherwise, if $\mathsf{corrosion} \neq \emptyset$ then $\mathsf{surrTrue}$ decreases in size (line 7).

These two facts gives us the termination of the recursion, since $\mathsf{surrTrue}$ cannot decrease indefinitely (as the model is finite) and so there will be a cycle in which $\mathsf{corrosion} = \emptyset$.

For the second recursion, we use a similar argument:

- During a cycle, if $\mathsf{corrosion} = \emptyset$ (line 14) then $\mathsf{badBorder} = \emptyset$ (line 17) and so the recursion terminates.
- Otherwise, if $\mathsf{corrosion} \neq \emptyset$ then $\mathsf{surrMaybe}$ decreases in size (line 15).

With the same argument as before we have that at some point $\mathsf{corrosion} = \emptyset$, and so the termination.

Correctness: To prove the correctness, we state a Lemma whose proof is postponed:

Lemma 4.2.25. *The followings hold for $s \in X$, $c \in C$ and $a \in A$:*

$$\begin{aligned} s \vDash_\top \mathcal{S}(p,q) &\iff s \vDash_\top p \quad \textit{and } \Pi_s^+\left(\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle\right) = \emptyset \\ c \vDash_\bot \mathcal{S}(p,q) &\iff c \vDash_\bot p \quad \textit{or } \Pi_c^+\left(\langle q, \bot\rangle, \langle p \vee q, \bot\rangle\right) \neq \emptyset \\ a \vDash_\bot \mathcal{S}(p,q) &\iff a \vDash_\bot p \quad \textit{or } \Pi_a\left(\langle q, \bot\rangle, \langle p \vee q, \bot\rangle\right) \neq \emptyset \end{aligned}$$

First of all, note that in the algorithm we set

$$\begin{aligned} &\mathsf{surrTrue} = [\![p]\!]_\top && \text{(line 1)} \\ &\mathsf{surrMaybe} = [\![p]\!]_? && \text{(line 2)} \\ &\mathsf{surrFalse} = [\![p]\!]_\bot && \text{(line 3)} \end{aligned}$$

so the three sets surrTrue, surrMaybe and surrFalse at first form a partition of the state space X. This property is preserved during all the algorithm, as we can check one assignment at a time:

- In lines 6-8, we take corrosion as a subset of surrTrue (line 6) and then we "move" its elements from surrTrue to surrMaybe (lines 7-8), thus maintaining the property.
- In lines 14-16, we take corrosion as a subset of surrMaybe (line 14) and then we "move" its elements from surrMaybe to surrFalse (lines 15-16), thus maintaining the property.

So we only need to show that $\mathsf{surrTrue} = [\![\mathcal{S}(p,q)]\!]_\top$ and $\mathsf{surrFalse} = [\![\mathcal{S}(p,q)]\!]_\bot$ to prove the correctness.

Case $\mathsf{surrTrue} = [\![\mathcal{S}(p,q)]\!]_\top$: we claim that during the first recursion (lines 5-10) we extracted from surrTrue *exactly* the states s such that $\Pi_s^+\left(\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle\right) \neq \emptyset$. Using Lemma 4.2.25 the result follows trivially.

We prove the claim by induction on the number of cycles of the recursion. In particular we want to show that at cycle i the set corrosion (which is extracted from surrTrue at line 7) contains exactly the points of $[\![p]\!]_\top$ for which there is a $\left(\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle\right)^+$-path of length *exactly* i. Of course, as $\mathsf{corrosion} \subseteq \mathsf{surrTrue} \subseteq [\![p]\!]_\top$ is true at each iteration we only have to check the path condition.

Notation 4.2.26. To improve readability we introduce the notation $\mathsf{corrosion}_i$ to indicate the set represented by the variable corrosion at iteration i (in particular at line 6).

In the first iteration, we have (expanding the definitions)

$$\begin{aligned} \mathsf{corrosion}_1 &= \mathrm{BN}([\![p]\!]_{\leq ?} \cap [\![p]\!]_{\leq ?}) \cap [\![p]\!]_\top \\ &= \mathrm{BN}([\![p \vee q]\!]_{\leq ?}) \cap [\![p]\!]_\top \end{aligned}$$

so exactly the points of $[\![p]\!]_\top$ for which there exists a $(\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle)^+$-path of length 1.
Now, suppose at iteration i we have

$$\mathsf{corrosion}_i = \left\{ s \in [\![p]\!]_\top \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle)^+\text{-path} \\ \text{of length exactly } i, \\ \text{but not shorter} \end{array} \right\}$$

By definition of $\mathsf{badBorder}_i$ (line 9) we have also that

$$\mathsf{badBorder}_i = \left\{ s \in [\![p]\!]_\top \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle)\text{-path} \\ \text{of length exactly } i, \\ \text{but not shorter} \end{array} \right\}$$

From this, at iteration $i+1$ we have

$$\begin{aligned} \mathsf{corrosion}_{i+1} &= \left\{ s \in [\![p]\!]_\top \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle)^+\text{-path} \\ \pi \text{ of length exactly } i+1 \\ \text{such that } \pi_0 \in \mathsf{surrTrue}_i, \\ \text{but not shorter} \end{array} \right\} \\ &= \left\{ s \in [\![p]\!]_\top \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \leq ?\rangle, \langle p \vee q, \leq ?\rangle)^+\text{-path} \\ \text{of length exactly } i+1, \\ \text{but not shorter} \end{array} \right\} \end{aligned}$$

where the last equality is easily provable.

Case $\mathsf{surrFalse} = [\![\mathcal{S}(p,q)]\!]_\bot$:

Notation 4.2.27. To simplify the notation we define the set $\mathsf{BC} = C \cup (A \cap [\![q]\!]_\bot)$. Note that this set is the same set as badCandidates in the algorithm.
As we already proved that $\mathsf{surrTrue} = [\![\mathcal{S}(p,q)]\!]_\top$, we now have only to show the second recursion "adjusts" the sets surrMaybe and surrFalse. In particular, we want to show that at each iteration $\mathsf{corrosion}_i$ is exactly the subset of $\mathsf{BC} \cap [\![p]\!]_{\geq ?}$ such that there exists a $(\langle q, \bot\rangle, \langle p \vee q, \bot\rangle)^+$-path of length exactly i. It's easy to verify using Lemma 4.2.25 that this implies the thesis (we are adding all these states to surrFalse at line 16).
In the first iteration, we have

$$\begin{aligned} \mathsf{corrosion}_1 &= \mathrm{BN}([\![p]\!]_\bot \cap [\![q]\!]_\bot) \cap \mathsf{BC} \cap [\![p]\!]_{\geq ?} \\ &= \mathrm{BN}([\![p \vee q]\!]_\bot) \cap \mathsf{BC} \cap [\![p]\!]_{\geq ?} \end{aligned}$$

so exactly the set of states of $\mathsf{BC} \cap [\![p]\!]_{\geq ?}$ for which there exists a $(\langle q, \bot\rangle, \langle p \vee q, \bot\rangle)^+$-path of length 1.
For the inductive step, we have

$$\mathsf{corrosion}_i = \left\{ s \in \mathsf{BC} \cap [\![p]\!]_{\geq ?} \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \bot\rangle, \langle p \vee q, \bot\rangle)^+\text{-path} \\ \text{of length exactly } i, \\ \text{but not shorter} \end{array} \right\}$$

and so by definition of $\mathsf{badBorder}_i$ (line 18)

$$\begin{aligned} \mathsf{badBorder}_i &= \left\{ s \in \mathsf{BC} \cap [\![p]\!]_{\geq ?} \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \bot\rangle, \langle p \vee q, \bot\rangle)\text{-path} \\ \text{of length exactly } i, \\ \text{but not shorter} \end{array} \right\} \\ &= \left\{ s \in [\![q]\!]_{\bot} \cap [\![p]\!]_{\geq ?} \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \bot\rangle, \langle p \vee q, \bot\rangle)\text{-path} \\ \text{of length exactly } i, \\ \text{but not shorter} \end{array} \right\} \end{aligned}$$

This means that at cycle $i+1$ we set $\mathsf{corrosion}_{i+1}$ as

$$\begin{aligned} \mathsf{corrosion}_{i+1} &= \left\{ s \in \mathsf{BC} \cap [\![p]\!]_{\geq ?} \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \bot\rangle, \langle p \vee q, \bot\rangle)^+\text{-path} \\ \pi \text{ of length exactly } i+1 \\ \text{such that} \\ \pi_0 \in \mathsf{badCandidates}_i, \\ \text{but not shorter} \end{array} \right\} \\ &= \left\{ s \in \mathsf{BC} \cap [\![p]\!]_{\geq ?} \;\middle|\; \begin{array}{l} \text{there exists a} \\ (\langle q, \bot\rangle, \langle p \vee q, \bot\rangle)^+\text{-path} \\ \text{of length exactly } i+1, \\ \text{but not shorter} \end{array} \right\} \end{aligned}$$

where the last equality is easily provable.

This concludes the proof of correctness.

Complexity: Of course, the complexity of this algorithm depends strongly on the actual implementation. Here we make use of the following facts:

Fact 4.2.28. *There is a data structure for sets such that:*

- *The operations $\cup$, $\cap$ and $\setminus$ are linear in the size of the second operand.*
- *The operation* BN *is linear in the number of back-edges of the argument.*

This gives us a way to estimate the complexity of each operation.

Algorithm 16: Complexity analysis. On the right an overestimate of the complexity of the single operations, up to multiplicative constants.

input : $\mathcal{M}$, $\mathcal{S}(p,q)$
output: $[\![\mathcal{S}(p,q)]\!]_\top$, $[\![\mathcal{S}(p,q)]\!]_?$, $[\![\mathcal{S}(p,q)]\!]_\bot$

```
1  surrTrue := [[p]]_⊤ ;                                          // |X|
2  surrMaybe := [[p]]_? ;                                         // |X|
3  surrFalse := [[p]]_⊥ ;                                         // |X|
4  badBorder := [[p]]_≤? ∩ [[q]]_≤? ;                             // |X|
5  repeat
6  |  corrosion := surrTrue ∩ BN(badBorder) ;                     // |BS(badBorder)|
7  |  surrTrue := surrTrue \ corrosion ;                          // |BS(badBorder)|
8  |  surrMaybe := surrMaybe ∪ corrosion ;                        // |BS(badBorder)|
9  |  badBorder := [[q]]_≤? ∩ corrosion ;                         // |BS(badBorder)|
10 until badBorder == ∅;
11 badBorder := [[p]]_⊥ ∩ [[q]]_⊥ ;                              // |X|
12 badCandidates := C ∪ (A ∩ [[q]]_⊥) ∩ surrMaybe ;              // |X|
13 repeat
14 |  corrosion := badCandidates ∩ BN(badBorder) ;                // |BS(badBorder)|
15 |  badCandidates := badCandidates \ corrosion ;                // |BS(badBorder)|
16 |  surrMaybe := surrMaybe \ corrosion ;                        // |BS(badBorder)|
17 |  surrFalse := surrFalse ∪ corrosion ;                        // |BS(badBorder)|
18 |  badBorder := corrosion ∩ [[q]]_⊥ ;                          // |BS(badBorder)|
19 until badBorder == ∅;
20 return surrTrue, surrMaybe, surrFalse
```

Now the thesis is easily proved by noticing that a state can enter the set badBorder at most once per recursion, and so the complexity of each recursion is bounded by $|{\to}| \geq |X|$.

□

Proof of Lemma 4.2.25: The only case to consider is when $s \vDash_\top \mathcal{S}(p,q)$ as the others follow from definition of the semantics.

To prove this case, we can carry the same proof as Lemma 3.1.24. In fact in the proof we only used that

$$[\![\neg p]\!] = \overline{[\![p]\!]}$$
$$[\![\neg q]\!] = \overline{[\![q]\!]}$$
$$[\![\neg p \wedge \neg q]\!] = \overline{[\![p \vee q]\!]}$$

so here we obtain the result from the equalities

$$\begin{aligned}\llbracket p \rrbracket_{\leq ?} &= \overline{\llbracket p \rrbracket_{\top}} \\ \llbracket q \rrbracket_{\leq ?} &= \overline{\llbracket q \rrbracket_{\top}} \\ \llbracket q \vee p \rrbracket_{\leq ?} &= \overline{\llbracket p \vee q \rrbracket_{\top}}\end{aligned}$$

□

As it will be useful later, we present here an alternative version of the algorithms (Algorithms 17 and 18) using a multi-agent approach. The idea behind this is that each state can be thought as an agent which has to *decide* if it entails the formula based on a *local communication* with the other agents in the system which are *near* it (meaning, adjacent states).

The main advantage of this approach is that the algorithm is easily implemented in a parallelized framework, for example using a map-reduction approach.

Moral 4.2.29. Algorithm 17 is a small modification of Algorithm 14. The main difference is the order in which the truth values are assigned.

Algorithm 18 is *exactly* the same as Algorithm 15, but with a different notation. In particular, in the pseudo-code states and sets are treated like *objects* and so they have properties and methods (accessing a property or method is indicated with the usual dot notation). The method flag contains the semantics of formulas evaluated at the given state (i.e., $s.\mathsf{flag}[\varphi] := \top$ means state s thinks that the formula φ has value $\top$).

In both recursions of Algorithm 18, at every iteration some states are processed (elements in toProcess) by sending a message along the edges of their backward star (edges in messages), which are then processed by the receiver (e.source in the algorithm). The stopping condition is that there are no more nodes to process, or equivalently that no messages have been sent.

4.2.3 Choosing Which States to Expand

Now we need only the last ingredient: which nodes do we expand at each iteration? As in the `CEGAR` case, the idea is to use the information we obtained from the model checking procedure for the abstract model.

So the idea is to inductively search the "reasons" why a sub-formula is not determined at a state. We now introduce a terminology we will use in the rest of the section (this is not meant to be a formal definition).

Notation 4.2.30.

Fix a state s and a formula φ such that $s \vDash_? \varphi$. We say that a pair state-formula $\langle t, \psi \rangle$ is a *reason* why $s \vDash_? \varphi$ (or simply a reason) if by knowing $t \vDash_\top \psi$ or $t \vDash_\bot \psi$ we gain more information to tell if $s \vDash_\top \varphi$ or $s \vDash_\bot \varphi$.

Algorithm 17: Alternative algorithm to compute $\mathcal{N}(p)$

```
input : M, N(p)
for s ∈ X do
    s.flag[N(p)] := ⊤ ;
for e ∈ FN([[p]]_{≥?}) do
    e.target.flag[N(p)] := ? ;
for s ∈ [[p]]_⊤ do
    if s ∈ A then
        s.flag[N(p)] := ⊤ ;
    else
        for e ∈ FN({s}) do
            e.target.flag[N(p)] := ⊤ ;
```

This suggests the following modifications of the model checking algorithms (Algorithms 19, 20 and 21):

Moral 4.2.31. The lines with an asterisk were added to the algorithms to compute the reasons. Note that the lines added don't affect the correctness (only properties reasons and clues are affected) and the complexity (every additional operation has constant time complexity).

Note also that in the $\mathcal{S}$ algorithm we included a property named clues to nodes. The reason is that in this case for each node there could be a huge number of *reasons* (comparable to $|X|$ for each node), and so we need to store implicitly this information to maintain a linear-time complexity when we choose which nodes to expand.

We state here a property of the algorithm that will be useful in what follows.

Lemma 4.2.32. *Fix a formula φ. Consider the relation*

$$sRt \iff t \in s.\mathit{clues}[\varphi]$$

Then this relation is acyclic.

Proof. The result is trivial except when $\varphi \equiv \mathcal{S}(\psi_1, \psi_2)$, so we'll examine only this case.

Consider the first recursion. In this proof we use the notation $\mathsf{toProcess}_i$ to indicate the set toProcess at the ith iteration of the recursion.

As proved before, a state s can enter the set toProcess at most one time during the entire recursion, so we assign to s the only value i such that $s \in \mathsf{toProcess}_i$. We will say that s *enters at stage* i and indicate it with $\mathtt{stage}(s) = i$.

Algorithm 18: Alternative algorithm to compute $\mathcal{S}(p,q)$

input : $\mathcal{M}$, $\mathcal{S}(p,q)$

```
1  for s ∈ [[p]]_⊤ do
2      s.flag[S(p,q)] := ⊤ ;
3  for s ∈ [[p]]_? do
4      s.flag[S(p,q)] := ? ;
5  for s ∈ [[p]]_⊥ do
6      s.flag[S(p,q)] := ⊥ ;
7  toProcess := {s ∈ X|(s.flag[p] ≤ ?)&&(s.flag[q] ≤ ?)} ;
8  repeat
9      messages := {e ∈ BS (toProcess) |e.source.flag[S(p,q)] == ⊤} ;
10     toProcess := ∅ ;
11     for e ∈ messages do
12         e.source.flag[S(p,q)] := ? ;
13         if e.source.flag[q] ≤ ? then
14             toProcess.add( e.source) ;
15 until toProcess == ∅;
16 toProcess := {s ∈ X|(s.flag[p] == ⊥)&&(s.flag[q] == ⊥)} ;
17 for s ∈ X do
18     if
         (s.flag[S(p,q)] == ?)&&((s.state == concrete)||(s.flag[q] == ⊥))
         then
19         s.badCandidate := ⊤ ;
20     else
21         s.badCandidate := ⊥ ;
22 repeat
23     messages := {e ∈ BS (toProcess) |e.source.badCandidate == ⊤} ;
24     toProcess := ∅ ;
25     for e ∈ messages do
26         e.source.badCandidate := ⊥ ;
27         e.source.flag[S(p,q)] := ⊥ ;
28         if e.source.flag[q] == ⊥ then
29             toProcess.add(e.source) ;
30 until toProcess == ∅;
```

Algorithm 19: Algorithm for boolean operators with *reasons*.

```
   input  : M, p
 1 for s ∈ X do
 2     if L(p, s) == ⊤ then
 3         s.flag[p] := ⊤ ;
 4     else if L(p, s) == ? then
 5         s.flag[p] := ? ;
 6         s.reasons[p].add( ⟨s, NULL⟩ ) ;                    // (*)
 7     else
 8         s.flag[p] := ⊥ ;
 9 return;

   input  : M, ¬p
10 for s ∈ X do
11     s.flag[¬p] := ¬s.flag[p] ;
12     if s.flag[¬p] == ? then                                // (*)
13         s.reasons[¬p].add( ⟨s, p⟩ ) ;                      // (*)
14 return;

   input  : M, p ∧ q
15 for s ∈ X do
16     s.flag[p ∧ q] := s.flag[p] ∧ s.flag[q] ;
17     if s.flag[p ∧ q] == ? then                             // (*)
18         if s.flag[p] == ? then                             // (*)
19             s.reasons[p ∧ q].add( ⟨s, p⟩ ) ;               // (*)
20         if s.flag[q] == ? then                             // (*)
21             s.reasons[p ∧ q].add( ⟨s, q⟩ ) ;               // (*)
22 return;
```

Now note the only line of code that modify the set clues is line 14 where we add e.target to e.source.clues. So we have:

$$\begin{aligned} sRt \iff & \text{ at a certain cycle, there exists } \mathsf{e} \in \mathsf{messages} \\ & \text{such that } s = \mathsf{e.source} \text{ and } t = \mathsf{e.target} \\ \iff & \text{ at a certain cycle, } t \in \mathsf{toProcess} \text{ and} \\ & s \to t \text{ and } s.\mathsf{flag}[\mathcal{S}(p,q)] = \top \end{aligned}$$

In particular, the last condition implies that s will enter the set toProcess in the next cycle, and so

$$sRt \implies \mathtt{stage}(s) > \mathtt{stage}(t)$$

from which follows the acyclicity.

□

Remark 4.2.33. Note that by examining the proof above we can show something more, namely that *every state* that entered the set toProcess (and so every state s such that $s \vDash_? \mathcal{S}(\psi_1, \psi_2)$) can reach a state t such that $\mathsf{stage}(t) = 0$ with an R-path.

Now that we added code to "take track of the reasons", we need an actual algorithm to choose which nodes to expand. We will define an algorithm recursive on the structure of the formula, namely Algorithm 22.

Lemma 4.2.34. *Algorithm 22 terminates and has linear-time complexity (over the encoding of the model).*

Proof.

Termination: In the algorithm there are two recursions: the recursive call of the function RecursiveReasons and a repeat-until recursion (lines $28-36$). We have to prove that both have a bounded number of cycles.

About the recursive call of RecursiveReasons, note that the second argument given to the function (the formula) is always a sub-formula of φ (lines 4, $12-13$, 18, $37-38$). So the result is easily proved by induction over φ given the base case terminates, but this is trivial to prove.

About the repeat-until recursion, note that $t \in \mathsf{cluesAccumulator}_{i+1}$ if and only if there exists $s \in \mathsf{cluesAccumulator}_i$ such that $t \in s.\mathsf{clues}[\mathcal{S}(\psi_1, \psi_2)]$. But by Lemma 4.2.32 we already know this relation is acyclic, and so it follows that a node could enter cluesAccumulator at most once during the recursion. This implies that at a certain point $\mathsf{cluesAccumulator} = \emptyset$ and so the termination of the recursion.

Complexity: It's easy to verify that the number of recursive calls of the function depends only on the formula φ, so what we have to show is that each case has linear time complexity in the encoding of the model.

- **Case** $\varphi \equiv p, \neg\psi$**:** the proof is trivial.
- **Case** $\varphi \equiv \psi_1 \wedge \psi_2$**:** we only need to show the accumulation process at lines 8-11 has linear-time complexity. But this is true as the reasons for $\psi_1 \wedge \psi_2$ are of the form $\langle t, \psi_1\rangle$ or $\langle t, \psi_2\rangle$, and so the number of reasons computed is at most twice the size of the model.
- **Case** $\varphi \equiv \mathcal{N}(\psi)$**:** we have to prove the accumulation process at line 17 has linear-time complexity. But this is easily done as the elements of $s.\mathsf{reasons}[\mathcal{N}(\psi)]$ have the form $\langle t, \psi\rangle$ for $t \in \mathrm{BN}(s)$, and so the number of reasons processed is bounded by the total number of edges of the model.
- **Case** $\varphi \equiv \mathcal{S}(\psi_1, \psi_2)$**:** first of all, note that every time the set $s.\mathsf{clues}[\mathcal{S}(\psi_1, \psi_2)]$ is accessed (lines 27 and 36) it is immediately set to $\emptyset$ (lines 28 and 37). This means a clue t can be processed at most $|\mathrm{FN}(t)|$ times.

  At lines 23-27 we accumulate reasons and clues stored in the set $\mathsf{maybeSet}$. This operation has linear-time complexity as the number of reasons is bounded by two times the number of nodes (they are of the form $\langle t, \psi_1\rangle$ and $\langle t, \psi_2\rangle$) and the number of clues is bounded by the number of edges (a clue for s is of the form t for $t \in \mathrm{BN}(s)$).

  The cycle at lines 28-36 has linear-time complexity, since
  - a clue t is processed at most $|\mathrm{FN}(t)|$ times, and so the operations at lines 32-37 are executed at most a linear number of times in the encoding of the model.
  - $s.\mathsf{reasons}[\mathcal{S}(\psi_1, \psi_2)]$ has at most two elements, as they are of the form $\langle s, \psi_1\rangle$ or $\langle s, \psi_2\rangle$ (check Algorithm 21).
  - the total time used on the operations at lines 30-31 and 36-37 is at most linear in the encoding of the model, since the time is bounded by a certain constant times the number of clue-processings.

  This implies the whole case has linear-time complexity.

□

Although we didn't focus so much on this aspect, a priori Algorithm 22 could not work as intended. The main problem is that the set given in output could be

- empty when there are states that don't decide the value of the formula.

- not empty when all states decide the value of the formula.

To show this doesn't happen we have the following:

Lemma 4.2.35. *Fix a formula φ and an* `MFM` $\mathcal{M}$.

1. *If $s \models_? \varphi$ then $s.\mathsf{reasons}[\varphi] \neq \emptyset$ or $s.\mathsf{clues}[\varphi] \neq \emptyset$ (meaning the sets computed in Algorithms 19, 20 and 21).*

2. *Algorithm 22 returns $\emptyset$ if and only if $[\![\varphi]\!]_? = \emptyset$.*

Proof.

1. The proof is trivial, except for Algorithm 21. In this case the result is easily proved using the value stage introduced in Lemma 4.2.32. In fact:

   - Each state s such that $s \models_? \mathcal{S}(p, q)$ enters the set $\mathsf{toProcess}$ and so the value $\mathsf{stage}(s)$ is defined.
   - $s \models_? \mathcal{S}(p, q)$ and $\mathsf{stage}(s) = 0$ means $s \models_? p$, and so $s.\mathsf{reasons} \neq \emptyset$ (lines $3 - 5$).
   - $\mathsf{stage}(s) \neq 0$ means $s.\mathsf{clues} \neq \emptyset$ (line 14).

2. Is trivial to show that if $[\![\varphi]\!]_? = \emptyset$ then the output is $\emptyset$, as the algorithm keeps calling the function $\mathsf{RecursiveReasons}$ with $\mathsf{maybeSet} = \emptyset$. So now we only need to show that if $[\![\varphi]\!]_? \neq \emptyset$ then the algorithm returns a non empty set.

   We can prove this result by induction over the structure of the formula φ given in input. Again, the only non trivial case is when $\varphi \equiv \mathcal{S}(\psi_1, \psi_2)$. Moreover, by induction hypothesis and examining lines $37 - 39$, we only need to prove that if $\mathsf{maybeSet} \neq \emptyset$, then $\mathsf{accumulatorPsiOne} \cup \mathsf{accumulatorPsiTwo} \neq \emptyset$.

   As we have stated in Remark 4.2.33 that

   - stage is defined on each state s such that $s \models_? \mathcal{S}(\psi_1, \psi_2)$.
   - For each state s such that stage is defined, there exists a path $\pi = (\pi_0 = s, \ldots, \pi_l)$ such that $\pi_{i+1} \in \pi_i.\mathsf{clues}[\mathcal{S}(\psi_1, \psi_2)]$ and $\mathsf{stage}(\pi_l) = \emptyset$.

   Now fix s and π as above and suppose by contradiction that for each $i \leq l$ it holds $\pi_i.\mathsf{reasons}[\mathcal{S}(\psi_1, \psi_2)] = \emptyset$. By analyzing lines $3 - 5$ and $9 - 18$ it means that:

   - For $0 < i \leq l$, $\pi_i \models_\perp \psi_2$ (lines $17 - 18$).
   - $\pi_l \models_\perp \psi_1$ (line 5).

And so we have that π is a $(\langle \psi_2, \bot \rangle, \langle \psi_1 \vee \psi_2, \bot \rangle)^+$, implying that $s \vDash_\bot \mathcal{S}(\psi_1, \psi_2)$. But this is absurd, as we already have that $s \vDash_? \mathcal{S}(\psi_1, \psi_2)$.

Now it's trivial to prove that if $s \in$ maybeSet, then Algorithm 22 will process each state of π and so will add the reasons of each π_i to accumulatorPsiOne or accumulatorPsiTwo, thus proving the thesis.

□

4.2.4 Conclusion

To conclude the chapter, we summarize the algorithm of abstraction-refinement in the spatial case.

So, given a $\mathcal{M}$, a chain of partitions $\overline{\mathcal{P}} = (\mathcal{P}_0, \ldots, \mathcal{P}_l)$ and a formula $\varphi \in$ SLCS, the algorithm is the following:

1. Compute the model $\mathcal{N} = \mathcal{M}_{\mathcal{P}_l}$.
2. Solve the model checking problem for $\langle \mathcal{N}, \varphi \rangle$ using Algorithms 19, 20 and 21.
3. If $[\![\varphi]\!]^{\mathcal{N}}_? = \emptyset$ then the algorithm is finished; otherwise choose the set of states to expand E using Algorithm 22.
4. Compute the model $\mathcal{N}'$ by expanding the states in E and repeat from point 2 using $\mathcal{N}'$ instead of $\mathcal{N}$.

The termination of each step of the algorithm was thoroughly analyzed, while the termination of the entire procedure is a direct consequence of the finiteness of the model and Lemma 4.2.35.

The correctness follows from Theorem 4.2.19 and the correctness of Algorithms 17 and 18.

The main point of this procedure is to solve the model checking problem *faster*, but we don't have yet significative experimental results to state the procedure actually works as intended. More results will be presented in the near future, with an actual implementation of the model checking algorithm above.

Algorithm 20: Algorithm to compute $\mathcal{N}(p)$ with *reasons*.

input : $\mathcal{M}$, $\mathcal{N}(p)$

```
1  for s ∈ X do
2      s.flag[N(p)] := ⊤ ;
3  for e ∈ FN([[p]]_{≥?}) do
4      e.target.flag[N(p)] := ? ;
5      e.target.reasons[N(p)].add( ⟨e.source, p⟩ ) ;          // (*)
6  for s ∈ [[p]]_⊤ do
7      if s ∈ A then
8          s.flag[N(p)] := ⊤ ;
9      else
10         for e ∈ FN({s}) do
11             e.target.flag[N(p)] := ⊤ ;
```

Algorithm 21: Algorithm to compute $\mathcal{S}(p,q)$ with reasons.

```
   input  : M, S(p,q)
 1 for s ∈ [[p]]_⊤ do
 2     s.flag[S(p,q)] := ⊤ ;
 3 for s ∈ [[p]]_? do
 4     s.flag[S(p,q)] := ? ;
 5     s.reasons[S(p,q)].add( ⟨s,p⟩ ) ;                          // (*)
 6 for s ∈ [[p]]_⊥ do
 7     s.flag[S(p,q)] := ⊥ ;
 8 toProcess := {s ∈ X|(s.flag[p] ≤ ?)&&(s.flag[q] ≤ ?)} ;
 9 repeat
10     messages := {e ∈ BS (toProcess) |e.source.flag[S(p,q)] == ⊤} ;
11     toProcess := ∅ ;
12     for e ∈ messages do
13         e.source.flag[S(p,q)] := ? ;
14         e.source.clues[S(p,q)].add(e.target) ;                // (*)
15         if e.source.flag[q] ≤ ? then
16             toProcess.add(e.source) ;
17             if e.source.flag[q] == ? then                     // (*)
18                 e.source.reasons[S(p,q)].add(⟨e.source, q⟩) ;  // (*)
19 until toProcess == ∅;
20 toProcess := {s ∈ X|(s.flag[p] == ⊥)&&(s.flag[q] == ⊥)} ;
21 for s ∈ X do
22     if
        (s.flag[S(p,q)] == ?)&&((s.state == concrete)||(s.flag[q] == ⊥))
       then
23         s.badCandidate := ⊤ ;
24     else
25         s.badCandidate := ⊥ ;
26 repeat
27     messages := {e ∈ BS (toProcess) |e.source.badCandidate == ⊤} ;
28     toProcess := ∅ ;
29     for e ∈ messages do
30         e.source.badCandidate := ⊥ ;
31         e.source.flag[S(p,q)] := ⊥ ;
32         e.source.reasons[S(p,q)] := ∅ ;                       // (*)
33         if e.source.flag[q] == ⊥ then
34             toProcess.add(e.source) ;
35 until toProcess == ∅;
```

Algorithm 22: Algorithm to choose the nodes to expand.

```
   input  : M, φ
   output: Nodes to expand.
1  maybeSet := {s ∈ X | s.flag[φ] == ?} ;
2  return RecursiveReasons(M, φ, maybeSet)

1  Function RecursiveReasons(M, φ, maybeSet)
2      switch φ do
3          case φ == p ∈ AP do return maybeSet;
4          case φ == ¬ψ do return RecursiveReasons(M, ψ, maybeSet);
5          case φ == ψ1 ∧ ψ2 do
6              accumulatorPsiOne := ∅ ;
7              accumulatorPsiTwo := ∅ ;
8              for s ∈ maybeSet do
9                  for ⟨t, χ⟩ ∈ s.reasons[φ] do
10                     if χ == ψ1 then accumulatorPsiOne.add(t);
11                     else accumulatorPsiTwo.add(t);
12             reasonsSetOne:=
                RecursiveReasons(M, ψ1, accumulatorPsiOne);
13             reasonsSetTwo:=
                RecursiveReasons(M, ψ2, accumulatorPsiTwo);
14             return reasonsSetOne ∪ reasonsSetTwo ;
15         case φ == N(ψ) do
16             accumulator := ∅ ;
17             for s ∈ accumulator do
                accumulator := accumulator ∪ s.reasons[N(ψ)];
18             return RecursiveReasons(M, ψ, accumulator);
19         case φ == S(ψ1, ψ2) do
20             accumulatorPsiOne := ∅ ;
21             accumulatorPsiTwo := ∅ ;
22             cluesAccumulator := ∅ ;
23             for s ∈ maybeSet do
24                 for ⟨t, χ⟩ ∈ s.reasons[S(ψ1, ψ2)] do
25                     if χ == ψ1 then accumulatorPsiOne.add(t);
26                     else accumulatorPsiTwo.add(t);
27                 cluesAccumulator := cluesAccumulator ∪ s.clues[S(ψ1, ψ2)] ;
28                 s.clues[S(ψ1, ψ2)] := ∅;
29             repeat
30                 cluesToProcess := cluesAccumulator ;
31                 cluesAccumulator := ∅ ;
32                 for s ∈ cluesToProcess do
33                     for ⟨t, χ⟩ ∈ s.reasons[S(ψ1, ψ2)] do
34                         if χ == ψ1 then accumulatorPsiOne.add(t);
35                         else accumulatorPsiTwo.add(t);
36                     cluesAccumulator := cluesAccumulator ∪ s.clues[S(ψ1, ψ2)];
37                     s.clues[S(ψ1, ψ2)] := ∅;
38             until cluesAccumulator == ∅;
39             reasonsSetOne:=
                RecursiveReasons(M, ψ1, accumulatorPsiOne);
40             reasonsSetTwo:=
                RecursiveReasons(M, ψ2, accumulatorPsiTwo);
41             return reasonsSetOne ∪ reasonsSetTwo ;
```

Bibliography

[1] Edmund M. Clarke. *The Birth of Model Checking*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[3] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement, 2000.

[5] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. Specifying and verifying properties of space - extended version. *CoRR*, abs/1406.6393, 2014.

[6] Vincenzo Ciancia, Gianluca Grilletti, Diego Latella, Michele Loreti, and Mieke Massink. *An Experimental Spatio-Temporal Model Checker*, pages 297–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[7] V. Ciancia, D. Latella, M. Massink, and R. Pakauskas. Exploring spatio-temporal properties of bike-sharing systems. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2015 IEEE International Conference on*, pages 74–79, Sept 2015.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[9] Arne Meier, Martin Mundhenk, Michael Thomas, and Heribert Vollmer. The complexity of satisfiability for fragments of {CTL} and ctl*. *Electronic Notes in Theoretical Computer Science*, 223:201 – 213, 2008. Proceedings of the Second Workshop on Reachability Problems in Computational Models (RP 2008).

[10] E. Allen Emerson. Model checking and the mu-calculus. In *DIMACS Series in Discrete Mathematics*, pages 185–214. American Mathematical Society, 1997.

[11] Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bull. Belg. Math. Soc. Simon Stevin*, 8(2):359–391, 2001.

[12] Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Information and Computation*, 205(8):1130 – 1148, 2007.

[13] Marco Aiello, Ian E. Pratt-Hartmann, and Johan F.A.K. van Benthem. *Handbook of Spatial Logics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[14] David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. In *PROCEEDINGS 3RD INTERNATIONAL CONFERENCE ON KNOWLEDGE REPRESENTATION AND REASONING*, 1992.

[15] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. *Spatial Logic and Spatial Model Checking for Closure Spaces*, pages 156–201. Springer International Publishing, 2016.

[16] Github - cherosene/ctl_logic: Model checker per la logica ctl.