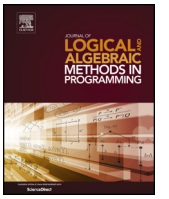
Journal of Logical and Algebraic Methods in Programming 120 (2021) 100642

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Verification of dynamic bisimulation theorems in Coq

Raul Fervari [a,b,*], Francisco Trucco [c], Beta Ziliani [a,b]

[a] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[b] FAMAF, Universidad Nacional de Córdoba, Argentina
[c] TU Wien, Austria

ARTICLE INFO

Article history:
Received 6 March 2020
Received in revised form 11 November 2020
Accepted 25 January 2021
Available online 27 January 2021

Keywords:
Modal logics
Dynamic logics
Bisimulation
Proof mechanization

ABSTRACT

Over the last years, the study of logics that can update a model while evaluating a formula has gained in interest. Motivated by many examples in practice such as hybrid logics, separation logics and dynamic epistemic logics, the ability to update a model has been investigated from a more general point of view. In this work, we formalize and verify in the proof assistant Coq, the bisimulation theorems for a particular family of dynamic logics that can change the structure of a relational model while evaluating a formula. Our framework covers update operators to perform different kinds of modifications on the accessibility relation, the valuation and the evaluation point of a model. The benefits of this formalization are twofold. First, our results apply for a wide variety of dynamic logics. Second, we argue that this is the first step towards the development of a modal logic library in Coq, which allows us to mechanize many relevant results.

1. Introduction

Historically, *modal logic* [1,2] has been understood as a logic to reason about different modes of truth. Under this perspective, it can be seen as an extension of propositional logic with *modalities*, that in specific contexts may have some particular interpretations. Examples of such interpretations are necessity, knowledge, belief, temporality, or obligation, to name a few. Nowadays, *modal logics* is a term defining a family of formalisms tailored to reasoning about relational structures, *i.e.*, to reasoning about graphs. This is a consequence of the insights provided by the most classical semantics for modal logics given in terms of the so-called Kripke structures [3]. In the wide spectrum of existing modal logics, one family has gained in interest lately: *dynamic modal logics*, *i.e.*, logics that can update the model while evaluating the truth of a formula. Some classic examples in this family are dynamic epistemic logics [4], separation logics [5,6], and hybrid logics [7], to mention a few.

However, the examples of dynamic logics mentioned above are specific instances designed with a particular goal in mind. Recently, several investigations focus on understanding the behavior of dynamic logics from a more general point of view (see *e.g.* [8–10]). Such a perspective enables us to investigate the properties of abstract operators that constitute the building blocks used to construct concrete modalities. Moreover, it allows us to obtain a general perspective of the impact of including and combining such kind of operators.

Nowadays, there is a vast literature investigating abstract update operators. The reactive Kripke semantics from [11,12] is one of the first attempts of modeling abstract updates. However, the dynamic information is kept inside the model instead

* Corresponding author.
E-mail addresses: rfervari@unc.edu.ar (R. Fervari), franciscoctrucco@gmail.com (F. Trucco), beta@mpi-sws.org (B. Ziliani).

https://doi.org/10.1016/j.jlamp.2021.100642
2352-2208/

of having operators performing actual modifications. An analysis of basic model modifiers is introduced in [13], in particular regarding expressivity and complexity. The logics studied therein combine different updates on the domain, the accessibility relation and the valuation of a model. Other works explore more specific instances of model updates. Domain modifications are covered, for instance with public announcement logics, whose expressivity and computational behavior are studied in [14,15]. Updates on the valuations are investigated from a dynamic epistemic logic perspective in [16]; more abstractly in the memory logics of [17–19] and hybrid logics of [20,7]; and recently from a game perspective in [21]. In most of these works, the operators are very expressive, and as a trade-off, they have high computational complexity.

The literature about operators that can change the accessibility relation of a model completes the picture of the dynamic modal logics family. Starting by the first ideas about sabotage logic presented in [8,9] (and later in [22]), various operations are presented in [23–25], more precisely, modalities to delete, add and swap-around an edge (both locally with respect to the current evaluation point, and globally in the whole model). These logics are called *relation-changing logics*, and the obtained results include expressivity and complexity. A more general approach is taken in [10], where the notion of 'updating a relation' is generalized, and some results can be proved for all the logics encompassed in this framework. For instance, a general notion of bisimulation is introduced, thus some properties can be proved for a whole family of logics by using this notion. This generality also allows us to extend the results to other kinds of updates; therefore we will use such an approach in this paper.

This is just a summary illustrating the importance of dynamic logics, and what is the kind of results that are explored in the literature. Moreover, all such results require proofs that are tedious and of high complexity, so it would be interesting to benefit of the use of computational tools in order to guide or verify (parts of) the proofs. Our work focuses on the formalization of a family of theorems for dynamic logics in the Coq proof assistant.

1.1. Computational assistance in complex proofs

The *Coq proof assistant* [26] is an interactive tool that helps us to perform complex mathematical proofs. It provides a formal language to formalize mathematical definitions, algorithms, theorems and their proofs. One of the main advantages of the Coq assistant is that it allows us to build mathematical proofs constructively. The underlying logic in Coq is an intuitionistic logic with dependent types, known as *the calculus of inductive constructions (CIC)*. Thanks to the Curry-Howard correspondence, propositions are interpreted as types and proofs are interpreted as programs with the type of the corresponding proposition. Thus, we can say that Coq is essentially a type verifier (see, *e.g.* [27]).

In the last years, several mathematical problems have been solved with the help of interactive tools like Coq; problems whose pen-and-paper proofs were put to doubt due to their high complexity. For instance, in [28], a problem from graph theory known as *the four color problem* was solved with the assistance of Coq. More recently, in [29] the *Kepler conjecture*, an open problem from combinatorial geometry, was proved by using a combination of the proof assistants HOL light [30] and Isabelle [31].

Motivated by examples like those in the last paragraph, we aim to develop a library to formalize and verify formal proofs in modal logic. In particular, we extend the formalization provided in [32] in order to model *dynamic logics*. Following [25, 10], we introduce a family of logics with dynamic operators that are parameterized by a *model update function*. A model update function takes a relational model as input, and returns a modified relational model. Then we formalize a bisimulation notion that is agnostic with respect to the model update function and mechanize the proof of two important theorems: the *invariance under bisimulation* theorem, and the Hennessy-Milner theorem. The invariance theorem establishes that given two models that are related by a bisimulation, they satisfy the same formulas of the corresponding language. The Hennessy-Milner theorem, states that under certain circumstances, the converse also holds. We consider this is the first step towards the development of a library for the mechanization of proofs for a wide variety of modal and dynamic logics.

Our choice of the Coq proof assistant over other tools is based on the following facts: 1) it is one of the most popular and actively developed proof assistants; 2) it is based on an intuitionistic logic, allowing us to clearly separate when we need to work in a classical setting, or when we can avoid resorting to classical axioms (not a real concern in this particular work, but that is important for studying non-classical modal logics like those in [33,34]; 3) it is the one we have the most expertise in; 4) it includes an impressive universe of tools to help coding proofs, making them look pretty much like pen-and-paper proofs (a good IDE, Unicode support, *etc.*).

1.2. Related work

There are several works exploring the mechanization of proofs for modal and non-classical logics. In [32], a formalization of the basic modal logic $\mathfrak{L}(\Diamond)$ is presented, in which we base our formalization. In addition, the author formalizes the extensions $S5$ and $S5^n$ [1], together with a natural deduction system. Then, the system is used to solve some logical puzzles. In [35] the modal logic cube, a family of extensions of the basic modal logic, is automatically verified using Isabelle/HOL. The main result is the proof of the inclusion relations between the cube's logics. A formalization in Coq of a Sahlqvists global correspondence theorem for the very simple Sahlqvist class is presented in [36]. Interestingly, from such formalization, it is possible to extract a verified Haskell program that computes correspondents of simple Sahlqvist formulas. Another approach has been taken in [37], in which a verification in Lean [38] of tableaux methods for modal logics is presented. In [39], Coq is used to verify the equivalence between two deductive systems for constructive modal logic.

Regarding non-classical logics, in [40], the authors present a formalization in Coq of *linear logic*, together with the mechanization of some theorems for such logic, such as a proof of cut-elimination. Finally, recent works present a proof language for *differential dynamic logic* [41] for applications in cyber-physical systems, in the theorem prover KeYmaera X [42].

It is worth to mention that the list of related work is obviously non-exhaustive, but it reflects an increasing interest in the use of proof assistants to mechanize proofs in the logic research area.

1.3. Contributions

This paper extends and greatly improves the results introduced in [43]. Therein, we presented a formalization in Coq of the so-called *relation-changing modal logics*, together with the mechanization of the *invariance under bisimulation theorem* from [10]. More precisely:

- We generalize the definition of model update function from [43] to cover a larger family of dynamic logics. Concretely, we allow updates not only in the evaluation point and the accessibility relation but also in the valuation of a relational model.
- We formalize a bisimulation notion for this family of logics and mechanize the proof of the *invariance under bisimulation theorem*. The theorem states that two bisimilar models satisfy the same formulas of the language.
- It is known that the converse of the invariance theorem does not always hold. However, as shown in [10], it holds under certain circumstances such as for finite models or for the so-called f-saturated models (the dynamic version of ω-saturated models). We formalize the notion of f-saturation and use it to prove in Coq that if two f-saturated models are modally equivalent, then they are bisimilar.
- In all cases, our mechanization permits languages defined with arbitrary sets of dynamic modalities. Thus, logics with multiple updates are covered in this framework. We include two of such logics as examples, namely *local sabotage logic* from [10] and *poison modal logic* from [21].

1.4. Organization

We start in Sec. 2 by providing a brief introduction to Coq's main features. In Sec. 3 we introduce the syntax, semantics and the notion of bisimulation for a family of dynamic modal logics. The formal definitions are accompanied by the corresponding formalization in Coq. Then, in Sec. 4 we present the mechanization of the different bisimulation-based theorems for this family of logics. We conclude in Sec. 5 with some final remarks and future lines of research.

2. A bird's view of Coq

A typical proof in Coq looks like the following:

```
Lemma and_intro: ∀(A B : Prop), A → B → A ∧ B.
Proof.
  move⇒ A B HA HB. split.
  – by apply HA.
  – by apply HB.
Qed.
```

This simple proof states that if you are given a proof of proposition `A` and another proof of proposition `B`, then you have a proof for their conjunction. In order to be able to state the lemma and prove it, Coq presents several domain-specific languages: Gallina, The Vernacular and several tactic languages. Below we provide a brief explanation of each of them.

Gallina: It is Coq's mathematical higher-level language and program specification language. In the example, the lemma's statement (what follows the `:`) is written in this language. Seen as a programming language, Gallina is a dependently-typed functional language, while seen as a logical system, Gallina is an *intuitionistic* higher-order type theory —in its purest form called *the calculus of inductive constructions* (CIC). Unlike other higher-order type theories like Isabelle [31], CIC embeds a notion of *pure* computations, which must be obviously terminating. Therefore, it embeds a notion of equality *up to computation*: we do not need to prove obvious facts such that 2 + 2 = 4. The restriction to pure computations stems from the fact that, if non-terminating computations were allowed, the logic would be unsound.

The Vernacular: It allows the definition of functions or predicates, the statement of mathematical theorems and software specifications, the machine checking of proofs and the extraction of certified programs to different languages. In the example we use the following Vernacular *commands*: `Lemma` indicates the desire to state a theorem; `Proof` starts the proof; `Qed` signals that the proof is completed, and therefore must be checked for errors and stored in the database of known facts if everything is correct. The reason for this check is to guarantee that the proof is indeed complete and that the tactics used to write the proof (see below) rightfully solved the problem.

Tactic languages: They allow us to perform and automate the deduction steps required to complete the proof. They are usually impure and might call external tools to help with the construction of the proof. What matters in the end is that they produce a *proof term* that can be checked by Coq's kernel *typechecker* (called at `Qed`).

In this example, we use two tactic languages: Ltac and Ssreflect, although several others exist. The former is Coq's standard (we do not require to load any specific library), while the latter must be imported with some variant of the `Require` Vernacular command.

In order to better understand how proofs work in Coq, we need to link them to the style of formal reasoning we employ as mathematicians. In essence, a proof is nothing but a tree of deduction rules, each of which relates a conclusion with a list of premises. There are two ways to understand a deduction rule: in *forward reasoning*, if we want to deduce the conclusion, we first try to deduce the list of premises and then use the deduction rule to prove the conclusion; in *backward reasoning*, we go in the opposite direction: in order to prove the conclusion, we must prove the premises. Tactics are typically deduction rules that implement backward reasoning: when applied to a conclusion, usually called a *goal*, a tactic replaces this goal with the *subgoals* it generates, one for each premise of the rule.

In the example, everything between `Proof` and `Qed` are tactic invocations. In a nutshell: `move⇒` is Ssreflect's tactic to introduce the formal parameters of the proof (propositional variables `A` and `B`, and the two hypotheses); Ltac's tactic `split` replaces the goal `A ∧B` with two subgoals, one for proving `A` and another for proving `B`; tactic `apply` is akin to applying a lemma or hypothesis to solve the goal, perhaps generating subgoals for its premises; and `by` is Ssreflect's *terminator* to ensure that the current goal is solved, or solving it in case it is trivial.

Not all tactics are as simple as `split`. For instance, the tactic `tauto` implements a decision procedure for intuitionistic propositional calculus, so it is appropriate to solve many trivial statements (actually, it is capable of solving the above lemma in just one tactic invocation). Ltac also allows the definition of complex user-defined tactics and decision procedures, although as a language it has some drawbacks: 1) its semantics are at times confusing and error-prone, and 2) it is essentially untyped, meaning that a typo can lead to obscure error messages when interpreting the proof. Therefore, we also use a third tactic language named Mtac2 [44], which is typed and has a well-defined semantics.

One key aspect that made Coq so popular is its interactive mode, in which a proof is processed sentence by sentence. Therefore, we can observe the changes performed to the goal at each step of the proof and act accordingly, without having to play the proof in our heads to understand what to do next. For instance, after executing the first sentence (that is, the one with `move⇒`), our goal changes to the following:

```
A : Prop
B : Prop
HA : A
HB : B
================
A ∧ B
```

Above the double line we have the hypotheses, whereas below the double line we have what we need to prove.

For an insightful description of Coq and its tactic languages the user is invited to read the official documentation [26] together with the specific documentation for each additional tactic language [45,44].

Before closing this section, we note that most of the work in creating and understanding a formalized proof is devoted to knowing how to codify a concept in the logic, and how to best solve the proofs. In the rest of the article, we devote considerable time to these two problems, explaining in detail each concept/tactic as we use them. We refer the reader to the code and comments in the code for details that are not core to the formalization.

3. Dynamic logics in Coq

In this section, we introduce the syntax, semantics, and the notion of bisimulation for a family of dynamic modal logics, together with their corresponding formalizations in Coq. The source code can be found at

https://github.com/liis-modal-logics/RelationChangingLogicsInCoq

As we will see below, it is easy to match the mathematical definitions with their counterpart in Coq.

3.1. Syntax

The syntax of the family of dynamic modal logics we introduce herein is a straightforward extension of the propositional logic. Let us introduce their syntax and semantics.

Definition 3.1 *(Syntax).* Let Prop be a countable, infinite set of propositional symbols. The set Form of formulas over Prop is defined as:

$$\mathsf{Form} ::= \bot \mid p \mid \varphi \Rightarrow \psi \mid \Diamond_i \varphi,$$

where $p \in \mathsf{Prop}$, $\Diamond_i \in \mathsf{Dyn}$ a set of dynamic operators, and $\varphi, \psi \in \mathsf{Form}$. Other operators are defined as usual, *e.g.*, $\neg\varphi$ is defined as $\varphi \Rightarrow \bot$, and $\Box_i \varphi$ is defined as $\neg\Diamond_i\neg\varphi$.

For $\mathcal{O} \subseteq \mathsf{Dyn}$ a set of dynamic operators, we call $\mathfrak{L}(\mathcal{O})$ the extension of the propositional language also allowing the operators in $\mathcal{O}$. If $\mathcal{O} = \{\Diamond_{i_1}, \ldots, \Diamond_{i_n}\}$, sometimes we write $\mathfrak{L}(\Diamond_{i_1}, \ldots, \Diamond_{i_n})$ instead of $\mathfrak{L}(\{\Diamond_{i_1}, \ldots, \Diamond_{i_n}\})$.

To formalize in Coq the syntax of dynamic modal logics, we first need to define the countable set of propositional symbols Prop. We can accomplish this by using an inductive type definition:

```
Inductive prop : Set := p : nat → prop.
```

This Vernacular command creates a new type called `prop` with a type constructor `p` that, given a natural number `n`, constructs an inhabitant of the type `prop`, namely `p n`. Clearly, `prop` correctly formalizes the countable infinite set Prop.

Before we define the syntax, we need to assume that a set of dynamic operators exists. This assumption is necessary because the definition of $\mathfrak{L}(\mathcal{O})$, with $\mathcal{O} \subseteq \mathsf{Dyn}$, depends on the existence of a set of dynamic operators Dyn.

```
Context (Dyn : Set).
```

Afterwards, when considering specific instantiations of logics, we will provide a specific value to `Dyn`. This is achieved using the module system of Coq, which is similar to that of the ML languages.[1]

Now we can give the definition of the set of formulas Form, as in Definition 3.1:

```
Inductive form : Set :=
 | Bottom : form
 | Atom  : prop → form
 | Impl  : form → form → form
 | DynDiam : Dyn → form → form.
```

Each line of this definition is interpreted as the members of the BNF from Definition 3.1. Other operators are defined as syntactic sugar. For example:

```
Definition Not (φ : form) : form := φ →' ⊥'.

Definition Top : form := ~'⊥'.

Definition And (φ ψ : form) : form := ~' (φ →' ~'ψ).

Definition Or (φ ψ : form) : form := ~'φ →' ψ.

Definition Iif (φ ψ : form) : form := (φ →' ψ) ∧' (ψ →' φ).

Definition DynBox (d : Dyn) (φ : form) : form := ~'◇ d ~'φ.
```

Note that in defining these formulas, we make use of Coq's facilities for writing and displaying code: its notation system and its support for Unicode. This way, we can write Coq code almost as in paper: `∧'` is a notation for `And`, `→'` for `Impl`, `◇` for `DynDiam`, and `~'` for `Not`. We append an apostrophe to those notations conflicting with Coq's own, as in `→'`.

3.2. Semantics

Semantically, formulas of $\mathfrak{L}(\mathcal{O})$ are evaluated in standard relational models.

Definition 3.2 *(Models).* A model is a triple $\mathfrak{M} = \langle W, R, V \rangle$, where W is the *domain*, a non-empty set whose elements are called points or states, $R \subseteq W \times W$ is the *accessibility relation*, and $V \subseteq \mathsf{Prop} \times W$ is the *valuation*.

Let w be a state in $\mathfrak{M}$, the pair $(\mathfrak{M}, w)$ is called a pointed model; we usually drop parentheses and call $\mathfrak{M}, w$ a pointed model.

[1] We will not discuss modules hereafter, and assume we can instantitate such variable when needed. The interested reader is invited to read the code and Coq's reference manual.

Notice that in standard presentations (see, *e.g.*, [1,2]) the valuation V in a model is a function $V : \mathsf{Prop} \to 2^W$. Instead, we chose an equivalent presentation given in terms of relations, which later will simplify some other definitions.

Also, in this article we restrict ourselves to models with only one accessibility relation. A generalization to models with multiple accessibility relations is possible, but leads to further choices concerning the definition of the dynamic operators (*e.g.*, which relation is affected by a given dynamic operator).

We will introduce operators that update the structure of a model. These operators will modify the accessibility relation and the valuation of the model. Changes in the domain are not considered, as will be discussed below.

Definition 3.3 *(Model update functions).* Given a domain W, a *model update function for* W is a function

$$f_W : W \times 2^{W^2} \times 2^{\mathsf{Prop} \times W} \to 2^{(W \times 2^{W^2} \times 2^{\mathsf{Prop} \times W})},$$

that takes a state in W, a binary relation over W and a valuation of symbols from Prop in W, and returns a set of possible updates to the state of evaluation, the accessibility relation and the valuation.

Let $\mathfrak{C}$ be a class of pointed models, a *family of model update functions* f is a class of model update functions, one for each domain of a model in $\mathfrak{C}$:

$$f = \{f_W \mid \langle W, R, V\rangle, w \in \mathfrak{C}\}.$$

A class $\mathfrak{C}$ is *closed under a family of model update functions* f if whenever $\langle W, R, V\rangle, w \in \mathfrak{C}$, then

$$\{\langle W, R', V'\rangle, v \mid f_W \in f,\ (v, R', V') \in f_W(w, R, V)\} \subseteq \mathfrak{C}.$$

Clearly, the class of all pointed models is closed under any family of model update functions. In this article, we only discuss the class of all models.

Notice, in the definition above, that a model update function is defined relative to a domain. We specifically require that all models with the same domain have the same model update function. This constraint limits the number of operators that can be captured in the framework, but at the same time leads to operators with more uniform behavior. We will discuss this issue further after we introduce the formal semantics of the dynamic operators below.

We proceed to formalize the concept of a relational model. In turn, this requires us to consider how powersets and relations are represented in Coq.

Let A be a set and `A` its corresponding formalization in Coq. In order to formalize a subset S of A (that is, $S \in 2^A$) we can view S as a function that for each element $a \in A$ determines whether a belongs to S or not. Naively, one could think that S can be modeled with a function from `A` to `bool`. However, this is overly restrictive, as it will force us to make S decidable (Coq's functions are guaranteed to terminate). Thus, we use the constructive type `Prop` instead of `bool` and write `A → Prop` to mean "a subset of A". Then, an element x is in S if Sx is a provable proposition. To help the readability of our definitions, we define a notation and write `set A` instead of `A → Prop`,[2] and we add a notation `x ∈ S` to mean `S x`.

For instance, if A is $\mathbb{N}$ and S is $\{x \mid x \text{ is odd}\}$, we can write in Coq:

```
Definition odd : set nat :=
  fix odd (n : nat) : Prop :=
    match n with
    | 0 ⇒ False (* False is the unprovable Prop *)
    | 1 ⇒ True (* True is the trivial Prop *)
    | S (S n') ⇒ odd n'
    end.
```

(What is in between `(*... *)` are comments in the code.)

Similarly, a binary relation R over A can be viewed as a function that given two elements $a, b \in A$ determines whether aRb or not. For this reason, we model R as `A → A → Prop`. Or, using Coq's standard library, simply `relation A`.

Now we are ready to introduce the formalization of the models of our logic. We can think a relational model as a triple consisting of a set W, a binary relation R defined over W, and a valuation function that for each element in W and each propositional symbol with type `prop`, decides whether that propositional symbol is valid or not in that element of W. In Coq we write the triple using a `Structure`:

```
Structure model := {
  m_states :> Set;
  m_rel: relation m_states;
  m_val: valuation m_states
}.
```

[2] In fact, the notation uses a definition from the standard library, `Ensembles A`, which is in turn defined as `A → Prop`.

The notation :> inserts a *coercion* from a model to a `Set`. This coercion acts like a simple overloading of a mathematical concept. This way, when having `m : model`, we can use `m` to mean both the model `m` or its set of states (`m.(m_states)`). In practice, every time a function requires a set of states we can pass `m` instead of `m.(m_states)`, and Gallina's type checker will take care of inserting the required coercion.

In the same way, as `m_states` defines a projector for the set W of states, `m_rel` and `m_val` define projectors for relation R and valuation V, respectively.

The type of the valuation deserves an explanation. The valuation is defined as `valuation m_states`, where `valuation` is defined as in Definition 3.2:

```
Definition valuation (W: Set) : Type := set (prop * W).
```

We will use two other structures: the self-explanatory `pointed_model` (which can be coerced to a `model`), and `state_model`, which essentially is like a pointed model, but in which the set `W` of states is parameterized. For it, we use a convenient notation using angle brackets. It is possible to coerce from a `pointed_model` to a `state_model`, and the other way around.

```
Structure pointed_model := {
  pm_model :> model;
  pm_point : pm_model
}.

Structure state_model (W: Set) := {
  st_point: W;
  st_rel: relation W;
  st_val: valuation W
}.

Notation "⟨ a , b , c ⟩" :=
  {| st_point := a; st_rel := b; st_val := c |}.
```

Before we can define satisfiability, we must provide a formalization for the type of all model update functions. Remember that a model update function takes a triple like our `state_model` and in essence returns a set of triples of the same type, for a given domain W. Therefore, we define the type of a model update function (`muf`) as:

```
Definition muf : Type := ∀(W : Set), state_model W → set (state_model W).
```

As with the mathematical definition f_W, a `muf` depends on the set W, and that is why we start with `∀ (W : Set), ...` .

Given that we have defined the notion of model update function and both the syntax and the models of the dynamic modal logics, we can now define the notion of satisfiability.

Definition 3.4 *(Semantics).* Let $\mathfrak{C}$ be a class of models, let $\mathfrak{M} = \langle W, R, V \rangle$ be a model with $w \in W$ a state such that $\mathfrak{M}, w \in \mathfrak{C}$. Let $\mathcal{O} \subseteq \mathsf{Dyn}$, and for each $\Diamond_f \in \mathcal{O}$, let f its associated family of model update functions on $\mathfrak{C}$. Let φ be a formula in $\mathfrak{L}(\mathcal{O})$. We say that $\mathfrak{M}, w$ *satisfies* φ, and write $\mathfrak{M}, w \models \varphi$, when

$$
\begin{array}{ll}
\mathfrak{M}, w \not\models \bot & \\
\mathfrak{M}, w \models p & \text{iff } (p, w) \in V \\
\mathfrak{M}, w \models \varphi \Rightarrow \psi & \text{iff } \mathfrak{M}, w \not\models \varphi \text{ or } \mathfrak{M}, w \models \psi \\
\mathfrak{M}, w \models \Diamond_f \varphi & \text{iff for some } (v, R', V') \in f_W(w, R, V),\ \langle W, R', V' \rangle, v \models \varphi.
\end{array}
$$

A formula φ is satisfiable if for some pointed model $\mathfrak{M}, w$ we have $\mathfrak{M}, w \models \varphi$. We write $\mathfrak{M}, w \equiv_{\mathfrak{L}(\mathcal{O})} \mathfrak{N}, v$ when both models satisfy the same $\mathfrak{L}(\mathcal{O})$-formulas, i.e., for all $\varphi \in \mathfrak{L}(\mathcal{O})$, $\mathfrak{M}, w \models \varphi$ if and only if $\mathfrak{N}, v \models \varphi$.

Notice, in the semantic definition, how a dynamic modal operator $\Diamond_f$ potentially changes the state of evaluation, the accessibility relation and the valuation. On the other hand, the domain remains the same, and hence all occurrences of each $\Diamond_f$ in a formula are evaluated using the same model update function f.

It must be clear at this point that to formalize the semantics, we need a function that assigns to each dynamic operator a model update function. We can assume that such a function exists, and we give it the name `F`.

```
Context (F : Dyn → muf).

Fixpoint satisfies (𝔐: pointed_model) (φ : form) : Prop :=
  match φ with
  | Bottom ⇒ False
  | Atom a ⇒ (a, 𝔐.(pm_point)) ∈ 𝔐.(m_val)
```

```
  | φ1 →' φ2 ⇒ (𝔐 ⊨ φ1) → (𝔐  ⊨ φ2)
  | ◇f φ ⇒
     let fw := F f 𝔐.(m_states) in
     ∃ p', p' ∈ fw 𝔐  ∧ p' ⊨ φ
  end
where "𝔐  ⊨ φ" := (satisfies 𝔐 φ).
```

Note how each case has a one-to-one correspondence with Definition 3.4. For readability, together with the recursive function `satisfies` we define the usual notation $\mathfrak{M} \models \varphi$. Also, note how Coq performs a coercion from model $\mathfrak{M}$ to the `state_model` expected by `fw`, and take the returned `state_model` $\mathfrak{M}'$ and coerce it into a `model` for the recursive call.

The definition of modal equivalence poses no challenges:

```
Definition equivalent (𝔐 𝔐': pointed_model) :=
  ∀ (φ: form), (𝔐 ⊨ φ) ↔ (𝔐' ⊨ φ).

Notation "𝔐  ≡  𝔐'" := (equivalent 𝔐 𝔐') (at level 0).
```

3.3. Examples of dynamic modal logics

First, notice that the classical modal diamond $\Diamond$ [1,2] is one particular instance of a dynamic operator, in which the accessibility relation and the valuation remain unchanged, and the evaluation state is changed by some successor via R. To simplify notation we use wv as a shorthand for $\{(w, v)\}$ or (w, v); context will always disambiguate the intended use. Let W be a domain, $w \in W$, $R \subseteq W^2$ and $V \subseteq \mathsf{Prop} \times W$, the model update function associated to $\Diamond$ is defined as

$$f_W^{\Diamond}(w, R, V) = \{(v, R, V) \mid wv \in R\}.$$

Such a model update function is defined in Coq as follows:

```
Definition diamond : muf :=
  fun W '⟨w, R, V⟩ '⟨v, R', V'⟩⇒
    R w v ∧ R = R' ∧ V = V'.
```

For a given pointed model with domain W, the model update function for $\Diamond$ returns a new pointed model (represented by a state, an accessibility relation and a valuation) which is exactly as the original one, but the evaluation point is changed. The new evaluation point is an R-successor of the original evaluation point. In the definition, the ' allows deconstructing the record into its components, in this case, using the notation provided for `state_model`.

Consider now the *local sabotage operator* from [10]. Given a binary relation R, let us introduce the notation $R^{-}_{wv} = R \backslash wv$. Define the model update function

$$f_W^{\mathsf{sb}}(w, R, V) = \{(v, R^{-}_{wv}, V) \mid wv \in R\},$$

that deletes the edge between the current evaluation point and some successor, and moves the evaluation to such succesor. The function defines the local sabotage operator $\Diamond_{\mathsf{sb}}$. In Coq, we first define the updated relation R^{-}_{wv} as `rel_minus`, which essentially states that it is false that w and v are related. Then, we define two modalities: the $\Diamond$ modality based on the previously defined `muf` called `diamond`, and the $\Diamond_{\mathsf{sb}}$ modality based on `rel_minus`. In this way we get the language $\mathfrak{L}(\Diamond, \Diamond_{\mathsf{sb}})$ by instantiating the type `Dyn` with a type containing the two operators, and function `F` that given one such operator returns the corresponding `muf`.

```
Definition rel_minus {W} (R: relation W) (w v: W) : relation W :=
  fun w' v'⇒
    (w = w' ∧ v = v' → False) ∨ (w ≠ w' ∨ v ≠ v' → R w' v').

Inductive Dyn := Diamond | Sb.

Definition F (f: Dyn) : muf :=
  match f with
  | Diamond ⇒ diamond
  | Sb ⇒ fun W '⟨w, R, V⟩ '⟨v, R', V'⟩⇒
    R w v ∧ R' = rel_minus R w v ∧ V' = V
  end.
```

As customary, we introduce some useful notation in the code:

```
Notation "◇  φ" := (DynDiam Diamond φ) (at level 65, right associativity).
Notation "'◇sb'  φ" := (DynDiam Sb φ) (at level 65, right associativity).
```

It is easy to show the logic $\mathfrak{L}(\Diamond, \Diamond_{sb})$ gains in expressivity with respect to $\mathfrak{L}(\Diamond)$. For example, the local sabotage operator $\Diamond_{sb}$ is logically stronger than the diamond operator when restricted to non-dynamic predicates, as the formula $\Diamond_{sb} p \to \Diamond p$ is valid. We state and prove such property in Coq:

```
Example valid_in_sb : ∀(p:prop) pm, pm ⊨ ◇sb p →' ◇p.
```

We direct the reader to the code for details of the proof. Also, $\Diamond_{sb}$ can force non-tree models (see, *e.g.*, [25,10]). For example, the formula $\Box_{sb}\Box\bot$ means that any local sabotage leads to a dead-end, hence the formula $\Diamond\Diamond\top \wedge \Box_{sb}\Box\bot$ can only be true at a reflexive state, a property that cannot be expressed in $\mathfrak{L}(\Diamond)$. Other relation-changing operators behave in a similar way (see [8,25,10] for details).

Finally, consider the *poison modal logic from* [21]. For a given 'poison atom' $p^\bullet$, the poison modality $\Diamond^\bullet$ is defined by the following model update function:

$$f^\bullet_W(w, R, V) = \{(v, R, V') \mid wv \in R \text{ and } V' = V \cup \{(p^\bullet, v)\}\}.$$

We start by formalizing the poison atom and a notation for it:

```
Context (poison_atom : prop).
Notation "p•" := poison_atom.
```

For the language $\mathfrak{L}(\Diamond, \Diamond^\bullet)$, we define the two modalities similarly as we did for sabotage.

```
Definition F (d: Dyn) : muf :=
  match d with
  | Diamond ⇒ diamond
  | Poison ⇒ fun W '⟨w, R, V⟩ '⟨v, R', V'⟩⇒
    R w v ∧ R' = R ∧ V' = (V ∪{{(p• , v)}})
end.
```

This logic is also very expressive. For instance, it can express that the current state has a reflexive edge with $\Diamond^\bullet p^\bullet$. This holds as long as $p^\bullet$ is initially false everywhere. The example `cycle` in the code illustrates this fact.

It is worth to notice that $\mathfrak{L}(\Diamond, \Diamond^\bullet)$ is closely related to memory logics (see, *e.g.*, [17,18]), and it is a fragment of some hybrid logics (see, *e.g.*, [7]). Also, other logics are easily encompassed with this approach, such graph modifiers [13], and public assignments [16], to name a few. This also illustrates the wide spectrum of logics that our framework covers.

3.4. Bisimulations

In modal model theory, the notion of bisimulation is a crucial tool. Typically, a bisimulation is a binary relation linking elements of the domains that have the same atomic information, and preserving the relational structure of the model. Because we also need to keep track of the changes on the accessibility relation and on the valuation that the dynamic operators may introduce, bisimulations are defined as relations that link triples consisting of the current state, accessibility relation and valuation. This follows the ideas introduced, *e.g.*, for relation-changing modal logics in [25,10], and for memory logics in [19]. Notice that the notion we introduce is parameterized with the corresponding model update functions, making the results general for the dynamic logics from Definition 3.1.

Definition 3.5 *(Bisimulations).* Let $\mathfrak{M} = \langle W, R, V\rangle$ and $\mathfrak{M}' = \langle W', R', V'\rangle$ be two models. Let $\mathcal{O} \subseteq \mathsf{Dyn}$, and let $\mathfrak{F}$ be the set of model update functions f such that $\Diamond_f \in \mathcal{O}$. A non-empty relation $Z \subseteq (W \times 2^{W^2} \times 2^{\mathsf{Prop}\times W}) \times (W' \times 2^{W'^2} \times 2^{\mathsf{Prop}\times W'})$ is an $\mathfrak{L}(\mathcal{O})$*-bisimulation* if it satisfies the following conditions. If $(w, S, U)Z(w', S', U')$ then for all $f \in \mathfrak{F}$,

(atomic harmony) for all $p \in \mathsf{Prop}$, $(p, w) \in U$ iff $(p, w') \in U'$;
(*f*_zig) if $(v, T, X) \in f_W(w, S, U)$, there is $(v', T', X') \in f_{W'}(w', S', U')$ s.t. $(v, T, X)Z(v', T', X')$;
(*f*_zag) if $(v', T', X') \in f_{W'}(w', S', U')$, there is $(v, T, X) \in f_W(w, S, U)$ s.t. $(v, T, X)Z(v', T', X')$.

Given two pointed models $\mathfrak{M}, w$ and $\mathfrak{M}', w'$, we say they are $\mathfrak{L}(\mathcal{O})$*-bisimilar* (notation, $\mathfrak{M}, w \leftrightarroweq_{\mathfrak{L}(\mathcal{O})} \mathfrak{M}', w'$) if there is an $\mathfrak{L}(\mathcal{O})$-bisimulation Z such that $(w, R, V)Z(w', R', V')$ where R and R' are the respective relations of $\mathfrak{M}$ and $\mathfrak{M}'$, and V, V' their respective valuations.

Summing up, the bisimulation notion for each logic $\mathfrak{L}(\mathcal{O})$ includes (atomic harmony) and the particular conditions for each model update function f such that $\Diamond_f \in \mathcal{O}$. For instance, according to the above definition, the (zig) and (zag) conditions for the basic modal logic $\mathfrak{L}(\Diamond)$ are defined as:

(zig) if $(w, v) \in S$, there is $v' \in W'$ s.t. $(w', v') \in S'$ and $(v, S, U)Z(v', S', U')$;
(zag) if $(w', v') \in S'$, there is $v \in W$ s.t. $(w, v) \in S$ and $(v, S, U)Z(v', S', U')$.

On the other hand, instantiating f with f^{sb} we get the following conditions:

(f^{sb}_zig) if $(w, v) \in S$, there is $v' \in W'$ s.t. $(w', v') \in S'$ and $(v, S^{-}_{wv}, U) Z (v', S'^{-}_{w'v'}, U')$;
(f^{sb}_zag) if $(w', v') \in S'$, there is $v \in W$ s.t. $(w, v) \in S$ and $(v, S^{-}_{wv}, U) Z (v', S'^{-}_{w'v'}, U')$.

Before formalizing the notion of bisimulation for these logics, let us define the type of relations between models. As introduced in Definition 3.5, the type of relations defining bisimulations relates triples of points, binary relations over the domains of the models, and valuations. This is precisely what a `state_model` is, as defined in Sec. 3.2.

```
Context {W W' : Set}.

Definition state_model_relation : Type :=
  state_model W → state_model W' → Prop.
```

The command `Context` allows us to state that we work under the assumption that we have two sets `W` and `W'`, an assumption we keep throughout the rest of the formalization. The curly braces around the variables' definitions indicate that they are *implicit*; that is, they can be inferred from the context in which the definitions are used.

To formalize the notion of bisimulation, we first define each condition separately and then use them as functions in the definition of bisimulation (`f_zag` clause is analogous). To state these conditions, we work under the assumption that we have a relation between models `Z`:

```
Context (Z : state_model_relation).

Definition atomic_harmony : Prop :=
  ∀ p p', Z p p' → ∀pr: prop,
    (pr, p.(st_point)) ∈ p.(st_val) ↔ (pr, p'.(st_point)) ∈ p'.(st_val).

Definition f_zig (f : muf) : Prop :=
  ∀ p q p', Z p p' →
    q ∈ f W p →
    (∃ q', q' ∈ f W' p' ∧ Z q q').
```

Each condition shares the precondition of bisimulation (see Definition 3.5), namely that `Z` relates models `p` and `p'`.

The notion of bisimulation is defined as:

```
Definition bisimulation : Prop :=
  atomic_harmony ∧ (∀ d, f_zig (F d)) ∧ (∀ d, f_zag (F d)).
```

Notice that we are asking the relation `Z` to satisfy `f_zig` and `f_zag` for all dynamic operators to be a bisimulation.

Two models are bisimilar if there exists a bisimulation `Z` relating the models (see Definition 3.5):

```
Definition bisimilar (𝔐 𝔐': pointed_model) : Prop :=
  ∃ Z, bisimulation Z ∧ Z 𝔐 𝔐'.

Notation "𝔐 ⇆ 𝔐'" := (bisimilar 𝔐 𝔐') (at level 30).
```

4. Formalizing bisimulation theorems

4.1. Invariance under bisimulation

In the rest of this section, we will present our formalization of the proof of the general invariance theorem. It establishes that the bisimulation notion captures the intended structural properties in order to preserve logical equivalence (over the appropriate language).

Theorem 4.1 *(Invariance). Let $\mathcal{O} \subseteq$ Dyn, and for each $\Diamond_f \in \mathcal{O}$ let f be its associated family of model update functions. Let $\mathfrak{M}, w$ and $\mathfrak{M}, w'$ be two pointed models. Then, $\mathfrak{M}, w \leftrightarrows_{\mathfrak{L}(\mathcal{O})} \mathfrak{M}', w'$ implies $\mathfrak{M}, w \equiv_{\mathfrak{L}(\mathcal{O})} \mathfrak{M}', w'$.*

Proof. Let $\mathfrak{M} = \langle W, R, V \rangle$ and $\mathfrak{M}' = \langle W', R', V' \rangle$ with $w \in W$ and $w' \in W'$, such that $\mathfrak{M}, w \leftrightarrows_{\mathfrak{L}(\mathcal{O})} \mathfrak{M}', w'$. Then there exists an $\mathfrak{L}(\mathcal{O})$-bisimulation Z such that $(w, R, V) Z (w', R', V')$.

We prove the theorem by structural induction on $\mathfrak{L}(\mathcal{O})$-formulas. In fact, we prove a more general result. Let $S \subseteq W^2$, $S' \subseteq W'^2$, $U \subseteq (\mathsf{Prop} \times W)$, $U' \subseteq (\mathsf{Prop} \times W')$ such that $(w, S, U) Z (w', S', U')$ for some $\mathfrak{L}(\mathcal{O})$-bisimulation Z, we will show that $\langle W, S, U \rangle, w \equiv_{\mathfrak{L}(\mathcal{O})} \langle W', S', U' \rangle, w'$.

[$\bot$ case:] Since $\bot$ is unsatisfiable, $\langle W, S, U\rangle, w \models \bot$ iff $\langle W', S', U'\rangle, w' \models \bot$.
[p case:]

$$\begin{aligned}
&\langle W, S, U\rangle, w \models p \\
\Leftrightarrow\ & (p, w) \in U && (\models) \\
\Leftrightarrow\ & (p, w') \in U' && \text{(atomic harmony)} \\
\Leftrightarrow\ & \langle W', S', U'\rangle, w' \models p && (\models)
\end{aligned}$$

[$\varphi \Rightarrow \psi$ case:]

$$\begin{aligned}
&\langle W, S, U\rangle, w \models \varphi \Rightarrow \psi \\
\Leftrightarrow\ & \langle W, S, U\rangle, w \not\models \varphi \text{ or } \langle W, S, U\rangle, w \models \psi && (\models) \\
\Leftrightarrow\ & \langle W', S', U'\rangle, w' \not\models \varphi \text{ or } \langle W', S', U'\rangle, w' \models \psi && (\text{ind. hypothesis}, \equiv_{\mathfrak{L}(\mathcal{O})}) \\
\Leftrightarrow\ & \langle W', S', U'\rangle, w' \models \varphi \Rightarrow \psi && (\models)
\end{aligned}$$

Let $\Diamond_f \in \mathcal{O}$, it remains to prove only a general case for $\Diamond_f \varphi$.
[$\Diamond_f \varphi$ case:] Suppose $\langle W, S, U\rangle, w \models \Diamond_f \varphi$. Then there is $(v, T, X) \in f_W(w, S, U)$ s.t. $\langle W, T, X\rangle, v \models \varphi$. Because Z is a bisimulation, by (f_zig) there exists $(v', T', X') \in f_{W'}(w', S', U')$ s.t. $(v, T, X) Z (v', T', X')$. By inductive hypothesis and definition of $\equiv_{\mathfrak{L}(\mathcal{O})}$, $\langle W', T', X'\rangle, v' \models \varphi$. Since $(v', T', X') \in f_{W'}(w', S', U')$, we get $\langle W', S', X'\rangle, w' \models \Diamond_f \varphi$. The other direction of the proof is analogous, using (f_zag) instead of (f_zig).

Therefore, since $(w, R, V) Z (w', R', V')$, we get $\mathfrak{M}, w \equiv_{\mathfrak{L}(\mathcal{O})} \mathfrak{M}', w'$. □

Now we are ready to formally state the Theorem of Invariance under Bisimulation (Theorem 4.1) in Coq:

```
Theorem InvarianceUnderBisimulation : ∀𝔐 𝔐' : pointed_model,
 𝔐 ⇆ 𝔐' → 𝔐 ≡ 𝔐'.
```

Proof The proof follows by structural induction on the formula φ. To get the right induction principle, first, we need to massage the goal and the list of hypotheses a bit so that φ is the first quantified variable. Therefore, the first lines of the proof are the following (between `(*comments *)` we explain their meaning):

```
move⇒ 𝔐 𝔐' bis φ. (* Put every variable in the context *)
move: 𝔐 𝔐' bis.   (* And bring everyone but φ back in the goal *)
```

At this point, the goal looks like

```
∀𝔐 𝔐' : pointed_model, 𝔐 ⇆ 𝔐' → 𝔐 ⊨ φ ↔ 𝔐' ⊨ φ
```

And our hypotheses look like follows:

```
Dyn : Set
F : Dyn → muf
φ : form
```

Now, we are ready to perform structural induction on φ:

```
elim: φ ⇒ [prop | | φ IHφ ψ IHψ | f φ IH] /=
          𝔐 𝔐'.
```

This line performs more than just structural induction, which is done with what is on the left of the ⇒. What comes after ⇒ is what is known as *intro patterns*: they permit several actions on the goal. In our case we name the different hypotheses in each case (`Atom`, `Bottom`, `Impl`, and `DynBox`, in that order) with `[prop | | φ IHφ ψ IHψ | f φ IH]`. Then, we simplify each generated sub-goal (`/=`), performing some computation on the goal, and re-introduce the models $\mathfrak{M}$ and $\mathfrak{M}'$ in the context.
The goal for the atomic case is

```
𝔐 ⇆ 𝔐'→
(prop, pm_point 𝔐) ∈ m_val 𝔐 ↔ (prop, pm_point 𝔐') ∈ m_val 𝔐'
```

Which is solved using Atomic Harmony (AH), from the assumption that the models are bisimilar.

```
move⇒ [Z [bis HZ]].
rewrite !to_st_val !to_st_point.
by apply ((get_AH bis) ?? HZ).
```

The first line assumes a model relation `Z` such that it is a bisimulation (`bis`), and in which relates the models (`HZ`). In the second line, we turn all the projections `pm_point` and `m_val` into projections of a `state_model`, as required by the definition of AH. Finally, the last line concludes by applying AH: `get_AH` takes AH from the hypothesis, and the notation `??` calls an Mtac2 tactic that inserts enough unknowns (`_` in Coq's lingo) until the whole expression is well-typed.

The bottom case is trivial (`by []`).

Now for the if case, we split the proof into two separate directions. First, we prove the left-to-right direction and then the right-to-left direction. This is performed using the `split` tactic. Luckily, we do not need to think about the two directions separately, since the same tactics work to prove both directions. That is why we end all the tactics in the following code with a semicolon (all but the last, of course). Both directions are proved simply by assuming the two antecedents `HIf` and `Hsat`, and then applying the inductive hypothesis for ψ and φ (using these two antecedents).

```
split; move⇒ HIf Hsat;
  apply (IHψ ?? bis);
  apply HIf;
  by apply (IHφ ?? bis).
```

The proof of the dynamic operator follows closely the same reasoning as the one presented above. First, like with the atom case, we assume the existence of a bisimulation `Z` (`move⇒[Z [bis HZ]]`). Second, like with the if case, we use the tactic `split` to consider both directions, although this time we prove them separately. We will explain the left-to-right direction only, as the other direction is analogous. After `split` Coq tells us that we need to prove the following:

```
(∃ p' : state_model 𝔐, p' ∈ F d 𝔐 𝔐 ∧ p' ⊨ φ) →
  ∃ p' : state_model 𝔐', p' ∈ F d 𝔐' 𝔐' ∧ p' ⊨ φ
```

First, we give a name to the existentially quantified values of the antecedent, together with its properties:

```
move⇒ [q [HqinfW Hsatq]].
```

We end up with the following context:

```
d : Dyn
φ : form
IH : ∀𝔐 𝔐' : pointed_model, 𝔐 ⇆ 𝔐' → 𝔐 ⊨ φ↔ 𝔐' ⊨ φ
𝔐, 𝔐' : pointed_model
Z : state_model_relation 𝔐 𝔐'
bis : bisimulation Z
HZ : Z 𝔐 𝔐'
q : state_model 𝔐
HqinfW : q ∈ F d 𝔐 𝔐
Hsatq : q ⊨ φ
```

At this point, we can apply the (f_zig) hypothesis from `bis` in the hypothesis `HqinfW`:

```
apply ((get_Zig bis) ?? HZ) in HqinfW
  as [q' [Hq'infW' HZqq']].
```

This introduces the following new hypotheses:

```
q' : state_model 𝔐'
Hq'infW' : q' ∈ F d 𝔐' 𝔐'
HZqq' : Z q q'
Hsatq : q ⊨ φ
```

Remember that our goal at this point is to prove that:

```
∃ p' : state_model 𝔐', p' ∈ F d 𝔐' 𝔐' ∧ p' ⊨ φ
```

The existential quantifier is removed by providing a witness:

```
∃ q'.
```

Now we only need to prove that:

```
q' ∈ F d 𝔐' 𝔐' ∧ q' ⊨ φ
```

The proposition on the left of the conjunction is identical to one of our hypotheses (namely `Hq'infW'`). The proposition on the right can be proved by applying the inductive hypothesis, although we need to massage the goal a bit to remove coercions. This is what we do in the following lines:

```
split; first by [].
apply (IH q) ; last by [].
∃ Z.
by rewrite !to_st_to_pm.
```

After analogously solving the right-to-left case the proof is ended, so we issue the closing command `Qed`.

4.2. Hennessy-Milner-style Theorem

The Invariance Theorem proves that every bisimulation defines an equivalence relation that is as least as fine as the one defined by modal equivalence. Over specific classes of models the two notions coincide. These classes are usually called Hennessy-Milner classes and the theorem stating the equivalence is called a Hennessy-Milner Theorem [46]. In this section, we generalize and formalize one kind of Hennessy-Milner class introduced in [10], and prove that over this class, model equivalence and bisimulations actually coincide.

A well-known result establishes that ω-saturated models are a Hennessy-Milner class for many modal languages (see [1] for details). We will define a suitable notion of ω-saturation for dynamic modal logics and prove a Hennessy-Milner Theorem with respect to the corresponding class of models.

Notice that all the definitions we will introduce generalize those presented in [10]. This is due to the fact that, in this paper, we work with a more general class of logics, since herein, we also allow updates in the valuation of the model. Thus, we need to adjust the notions of saturation and remake the proof accordingly.

Definition 4.2 *(f-saturation; $\mathfrak{F}$-saturation).* Let $\mathfrak{M} = \langle W, R, V\rangle$ be a model, and let $\mathfrak{S} \subseteq W \times 2^{W^2} \times 2^{(\mathsf{Prop}\times W)}$. Let Σ be a set of $\mathfrak{L}(\mathcal{O})$-formulas, for some $\mathcal{O} \subseteq \mathsf{Dyn}$. Σ is *satisfiable over* $\mathfrak{S}$ *in* $\mathfrak{M}$ if there is some $(u, S, X) \in \mathfrak{S}$ such that $\langle W, S, X\rangle, u \models \varphi$, for all $\varphi \in \Sigma$ (we will not mention $\mathfrak{M}$ when it is evident from context). Σ is *finitely satisfiable over* $\mathfrak{S}$ *in* $\mathfrak{M}$ if each finite subset of Σ is satisfiable over $\mathfrak{S}$.

Let $\mathfrak{F}$ be a set of families of model update functions in one-to-one correspondence with $\mathcal{O}$, and let $f \in \mathfrak{F}$. We say that $\mathfrak{M} = \langle W, R, V\rangle$ is *f-saturated* if for all Σ, and for all $(v, S, X) \in \bigcup_{g\in\mathfrak{F}} \mathsf{Img}(g_W) \cup \{(w, R, V) \mid w \in W\}$ whenever Σ is finitely satisfiable over $\mathfrak{S} = \{(t, T, U) \mid (t, T, U) \in f_W(v, S, X)\}$ then it is satisfiable over $\mathfrak{S}$.

A model $\mathfrak{M}$ is *$\mathfrak{F}$-saturated* if it is f-saturated, for all $f \in \mathfrak{F}$.

The definition of $\mathfrak{F}$-saturation is a variation of the standard definition of ω-saturation and intuitively requires ω-saturation in each possible updated model via every possible function from $\mathfrak{F}$, and also with respect to the set of possible model updates in each state via every function from $\mathfrak{F}$.

The notion of a set of formulas Σ being *satisfiable over* $\mathfrak{S}$ *in* $\mathfrak{M}$ is simply formalized as:

```
Section Satisfiability.

Context (𝔐 : model).
Context (𝔖 : set (state_model 𝔐)).
Context (Σ : set form).

Definition satisfiable :=
  ∃ st : state_model 𝔐,
    st ∈ 𝔖 ∧ (∀ φ: form, φ∈ Σ → st ⊨ φ).
```

The first lines create a *section* named `Satisfiability`, which allows us to specify local hypotheses for the definitions within the section. Once the section is closed (with the command `End Satisfiability`), these variables will be prepended as universally quantified to each definition within the section.

The definition of *finitely satisfiable* is defined likewise:

```
Definition finitely_satisfiable := ∀Δ: finset Σ,
  ∃ st : state_model 𝔐, st ∈ 𝔖 ∧
    Forall (fun φ: form⇒ st ⊨ φ) Δ.
```

Where the type `finset Σ` can be thought of as a (finite) list of formulas from Σ. The function `Forall P l` asserts that property `P` holds for each element in list `l`, in this case, that every formula in Δ is satisfiable.

As for the concepts of f-saturated and $\mathfrak{F}$-saturated, the latter called *saturated* in the code, are defined as follows:

```
Definition image_iden : set (state_model 𝔐) :=
  fun st ⇒ st_rel st = m_rel 𝔐 ∧ st_val st = m_val 𝔐.

Definition image_fw f : set (state_model 𝔐) :=
  fun st ⇒ ∃st': state_model 𝔐, st ∈ F f 𝔐 st'.

Definition image_Ufw : set (state_model 𝔐) :=
  fun st ⇒ ∃f, st ∈ image_fw f.

Definition image := image_iden ∪image_Ufw.

Definition f_saturated f :=
  ∀(Σ: set form)(st: state_model 𝔐),
    st ∈ image → let 𝔖 := F f 𝔐 st in
    finitely_satisfiable 𝔖 Σ→ satisfiable 𝔖 Σ.

Definition saturated := ∀f, f_saturated f.
```

Let us consider these definitions in reverse order. With `saturated` we state that a model is `f_saturated` for every model update function `f` considered. The definition of `f_saturated` is in perfect correspondence with the definition given above, although we need to explain how we compute the `image`: it is computed from the union of `image_iden`, which returns the set of the `state_models` with the same relation and valuation as $\mathfrak{M}$; and `image_Ufw`, which is the union of the images of every operator `f`.

Proposition 4.3. *Let $\mathcal{O} \subseteq$ Dyn and let $\mathfrak{F}$ be a set of families of model update functions in one-to-one correspondence with $\mathcal{O}$. Let $\mathfrak{M}, w, \mathfrak{M}', w'$ be two $\mathfrak{F}$-saturated models. Then,*

$$\mathfrak{M}, w \equiv_{\mathfrak{L}(\mathcal{O})} \mathfrak{M}', w' \text{ implies } \mathfrak{M}, w \leftrightarrows_{\mathfrak{L}(\mathcal{O})} \mathfrak{M}', w'.$$

Proof. We prove that when two $\mathfrak{F}$-saturated pointed models satisfy the same formulas, they are bisimilar.

Let $\mathfrak{M} = \langle W, R, V\rangle$ and $\mathfrak{M}' = \langle W', R', V'\rangle$ be given, and let $\mathfrak{F}$ be a set of families of model update functions in one-to-one correspondence with $\mathcal{O}$. Define the relation $\leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})}$ over $\bigcup_{f\in\mathfrak{F}} \mathsf{Img}(f_W) \cup \{(w, R, V)\} \times \bigcup_{f\in\mathfrak{F}} \mathsf{Img}(f_{W'}) \cup \{(w', R', V')\}$ such that:

$$(v, S, X) \leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})} (v', S', X') \text{ holds iff } \langle W, S, X\rangle, v \equiv_{\mathfrak{L}(\mathcal{O})} \langle W', S', X'\rangle, v'.$$

Notice that (by hypothesis) $(w, R, V) \leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})} (w', R', V')$. We show that $\leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})}$ is an $\mathfrak{L}(\mathcal{O})$-bisimulation.

Suppose $(s, S, X) \leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})} (s', S', X')$; we need to prove that each condition from Definition 3.5 holds.

We start by proving the (atomic harmony) condition. Let $p \in \mathsf{Prop}$, suppose that $(p, s) \in X$. Then, we have (by $\models$) $\langle W, S, X\rangle, s \models p$. By definition of $\leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})}$ and the hypothesis, $\langle W', S', X'\rangle, s' \models p$. Therefore, by $\models$, $(p, s') \in X'$, as wanted. Notice that each step works in both directions.

Now, we need to prove the (f_zig) and (f_zag) conditions for each $f \in \mathfrak{F}$.

Let $f \in \mathfrak{F}$. For the (f_zig) condition, let $(t, T, Y) \in f_W(s, S, X)$. We should prove that there is $(t', T', Y') \in f_W(s', S', X')$ such that $(t, T, Y) \leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})} (t', T', Y')$.

Let $\Sigma = \{\varphi \mid \langle W, T, Y\rangle, t \models \varphi\}$, for every $\Delta \subseteq_{\mathsf{fin}} \Sigma$ we have

$$\langle W, S, X\rangle, s \models \Diamond_f \bigwedge \Delta.$$

Notice that Δ can be $\emptyset$, thus as a convention we take $\bigwedge \emptyset$ as $\top$. By $\leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})}$, $\langle W', S', X'\rangle, s' \models \Diamond_f \bigwedge \Delta$, then (by $\models$) there exists $(t'_\Delta, T'_\Delta, Y'_\Delta) \in f_{W'}(s', S', X')$ such that $\langle W', T'_\Delta, Y'_\Delta\rangle, t'_\Delta \models \bigwedge \Delta$. Since this holds for each $\Delta \subseteq_{\mathsf{fin}} \Sigma$, we have that Σ is finitely satisfiable over

$$\mathfrak{S}' = \{(t'_\Delta, T'_\Delta, Y'_\Delta) \in f_{W'}(s', S', X') \mid \text{there is } \Delta \subseteq_{\mathsf{fin}} \Sigma, \text{ such that } \langle W', T'_\Delta, Y_\Delta\rangle, t'_\Delta \models \bigwedge \Delta\}.$$

As $\mathfrak{S}' \subseteq \mathfrak{S}'' = \{(t'', T'', Y'') \in f_{W'}(s', S', X')\}$, it follows that Σ is finitely satisfiable over $\mathfrak{S}''$. Hence, by $\mathfrak{F}$-saturation (and consequently, f-saturation) of $\mathfrak{M}'$, Σ is satisfiable over $\mathfrak{S}''$, *i.e.*, there exists $(t', T', Y') \in \mathfrak{S}''$ satisfying Σ. Therefore we have $(t, T, Y) \leftrightsquigarrow_{\mathfrak{L}(\mathcal{O})} (t', T', Y')$.

The (f_zag) condition is proved by using similar steps. □

The formalization of the proof mimics exactly the steps just described. Here, we focus on the main definitions and direct the reader to the code for details.

The proof of the theorem is the corollary of a lemma stating that a new relation, called $\leftrightsquigarrow$ in the code, is a bisimulation. We start by defining $\leftrightsquigarrow$:

```
Definition equiv_in_image st st' :=
  st ∈ image 𝔐 ∧
  st' ∈ image 𝔐' ∧
  st ≡ st'.

Notation "a ↭ b" := (equiv_in_image a b) (at level 40).
```

Note how we restrict the domain of $\leftrightsquigarrow$ within its definition re-using the definition of `image` provided before.

The lemma is named in the code as:

```
Lemma equiv_in_image_bisimulation : bisimulation equiv_in_image.
```

And it resorts on the models being saturated:

```
Context (M_sat : saturated 𝔐).
Context (M'_sat : saturated 𝔐').
```

The proof follows the text closely: first proving (atomic harmony), and then (f_zig) and (f_zag). We omit the proof of the former, since it poses no challenges. As for (f_zig) and (f_zag), in the code, we decided to work step by step, even if that implies making the proof larger than strictly needed.

For (f_zig), we work under the following assumptions:

```
imgS : ⟨ s, S, X ⟩∈ image 𝔐
imgS' : ⟨ s', S', X' ⟩∈ image 𝔐'
SeqS' : (⟨ s, S, X ⟩) ≡ (⟨ s', S', X' ⟩)
tTYinsSX : ⟨ t, T, Y ⟩∈ D.F f 𝔐 ⟨ s, S, X ⟩
```

And we have to prove that:

```
∃ q' : state_model 𝔐',
  q' ∈ F f 𝔐' ⟨ s', S', X' ⟩ ∧ ⟨ t, T, Y ⟩↭ q'
```

We defined the set of formulas Σ that are satisfied by $\langle\, t\,,\, T\,,\, Y\,\rangle$, and prove that any subset Δ of Σ is also satisfied, with their formulas conjoined in a *big conjunction* ($\bigwedge\Delta$):

```
pose Σ: set form := (fun φ⇒ ⟨ t , T , Y ⟩⊨ φ).

have sat_big_and :
  ∀Δ : finset Σ, ⟨ t, T, Y⟩ ⊨ ⋀Δ.
```

The proof of `sat_big_and` is by a simple induction on Δ. By hypotheses and the just proved property we easily get:

```
have sat_next_big_and' :
  ∀Δ : finset Σ, ∃st', st' ∈ F f 𝔐' ⟨s', S', X'⟩ ∧ st' ⊨ ⋀Δ.
```

Which allows us to conclude that $\mathfrak{S}'$ is finitely satisfiable:

```
pose 𝔖' : set (state_model _) :=
   fun st' ⇒ st' ∈ F f 𝔐' ⟨ s', S', X' ⟩∧
         ∃Δ : finset Σ, st' ⊨ ⋀Δ.

have 𝔖'_fsat : finitely_satisfiable 𝔖' Σ.
```

Again, we omit the details of the proof, which can be found in the files. We then generalize the result for $\mathfrak{S}''$, where we use the fact that our models are saturated:

```
pose 𝔖'' := F f 𝔐' ⟨ s', S', X' ⟩.
have 𝔖''_fsat : finitely_satisfiable 𝔖'' Σ.
  …
have 𝔖''_sat : satisfiable 𝔖'' Σ
  by apply M'_sat.
```

Thanks to $\mathfrak{S}''$ `_sat` we can now obtain the witness we need:

```
case: 𝔊"_sat ⇒ [ [t' T' Y'] [inS H] ].
∃⟨ t', T', Y' ⟩.
```

After some minor nuisances, we need to prove (for instance, that the witness is in the `image` of the model), we get to the point where we need to prove that:

```
(⟨ t, T, Y ⟩) ≡ (⟨ t', T', Y' ⟩)
```

This is mostly trivial, except that the case

```
⟨ t', T', Y' ⟩ ⊨ φ → ⟨ t, T, Y ⟩⊨ φ
```

for some formula φ requires classical reasoning, namely, that either a formula or its negation is satisfiable. We perform a case analysis on ⟨ t, T, Y ⟩ ⊨φ or ⟨ t, T, Y ⟩ ⊨~'φ being provable. The first case is trivial, since its precisely what we need to prove.

```
case: (sat_classic ⟨t, T, Y ⟩φ); first by [].
```

For the second case, we have to prove that

```
⟨ t, T, Y ⟩⊨ (~' φ) → ⟨t', T', Y' ⟩ ⊨ φ→ ⟨t, T, Y ⟩⊨ φ
```

under the following assumptions:

```
Σ := (fun φ: form ⇒ ⟨t, T, Y ⟩⊨ φ) : set (form)
H : ∀φ  : form, φ∈ Σ → ⟨t', T', Y' ⟩ ⊨ φ
```

Here it is easy to see the contradiction: ~' φ is in the set Σ, since it is entailed by ⟨ t, T, Y ⟩, but by `H` everything in Σ is also entailed by ⟨ t', T', Y' ⟩. Therefore, ⟨ t', T', Y' ⟩ ⊨ ~'φ, but we have as an hypothesis that ⟨ t', T', Y' ⟩ ⊨φ, so we get the contradiction. This reasoning is performed in the following lines:

```
fold (Σ (~' φ)).
move/H. apply2. simpl.
contradiction.
```

The first line replaces ⟨ t, T, Y ⟩ ⊨(~' φ) by its definitionally equivalent Σ (~' φ). The second line uses this assumption to apply it to `H`, obtaining the goal

```
⟨ t', T', Y' ⟩ ⊨ (~' φ) → ⟨t', T', Y' ⟩ ⊨ φ→ ⟨t, T, Y ⟩⊨ φ
```

Now we can apply the second assumption to the first and obtain `False`. For this, we call an Mtac2 tactic called `apply2`.

As usual, the proof of (*f*_zag) is almost identical.

5. Final remarks

In this work, we use the interactive proof assistant Coq to formalize the syntax, semantics and a notion of bisimulation for a family of dynamic logics. These logics contain modalities that update the evaluation point, the accessibility relation and the valuation of the model while evaluating a formula. One particularity of our formalization is that, following the definition from [10], dynamic modalities are parameterized by a *model update function*, *i.e.* a function that given a pointed model, returns an updated pointed model. In [10] model update functions are able to modify the evaluation point and the relation of a model, whereas herein, we generalize these functions to update also the valuation of a model. Thus, the definitions in Coq are simple but powerful enough to encompass a whole family of logics. With these definitions at hand, we recreated the proof of the *invariance under bisimulation theorem:* if two models are bisimilar for a determined logic, then they satisfy the same formulas. In addition, we show that for the class of models that are saturated by the model update functions involved in the language, the converse also holds: if two models are modally equivalent then they are bisimilar. Again, given the generality of our framework, each theorem only needs to be proved once, and they hold for any instance of model update functions. Examples of logics encompassed in this framework are sabotage logics [8,47], swap logic [24], arrow update logics [48,49], poison modal logic [21], the program-based relation-changing operations from [50–53] (which are as expressive as *propositional dynamic logic* [54] and whose bisimulation notion is the same as for the basic modal logic [1]), among others (*e.g.* [16,13,19]). Our choice of Coq as the proof assistant for the formalizations relies mainly on the following facts: it is one of the most popular and actively developed proof assistants; and, it includes an impressive universe of tools to help coding proofs, making them look pretty much like pen-and-paper proofs (a good IDE, Unicode support, etc.), which makes it easy to use and understand.

There are several interesting directions that we would like to explore in the future. The next step will be the mechanization of more complex results for dynamic logics, such as the correctness of the encodings into first-order and second-order

logic [10]. Some of these results can be proved for particular instances of model updates functions, while others can be proved for the general case. With these results we will complete the mechanization of the basic model theory for this family of dynamic logics, allowing us to go further and prove more complex results, such as the dynamic version of van Benthem's characterization theorem [55]. Moreover, we would like to use our framework for mechanizing proofs for more concrete instances of this family of logics, such as *dynamic epistemic logics* [4] and *modal separation logics* [56–59].

We consider that this work represents the first step towards a more serious use of an interactive proof assistant for proofs in modal logic. One of the main problems of existing formalizations of modal logics is that they all are too heterogeneous, and it makes it difficult to reuse previous results. Our main goal is the development of a modal logic library that allows us the mechanization in a uniform way of a wide variety of results in modal and dynamic logics.

Declaration of competing interest

No conflict of interest.

Acknowledgements

We would like to thank the three anonymous reviewers for their helpful comments and suggestions. This work is supported by projects ANPCyT-PICTs-2017-1130 and 2016-0215, Stic-AmSud 20-STIC-03 "DyLo-MPC", SECyT-UNC, GRFT Mincyt-Cba, and by the Laboratoire International Associé SINFIN.

References

[1] P. Blackburn, M. de Rijke, Y. Venema, Modal Logic, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2001.
[2] P. Blackburn, J. van Benthem, Modal logic: a semantic perspective, in: Handbook of Modal Logic, Elsevier, 2007, pp. 1–84.
[3] S. Kripke, Semantical analysis of modal logic I. normal propositional calculi, Zeitschrift fur mathematische Logik und Grundlagen der Mathematik 9 (1963) 67–96.
[4] H. van Ditmarsch, W. van der Hoek, B. Kooi, Dynamic Epistemic Logic, Synthese Library, Springer, 2007.
[5] J. Reynolds, Separation logic: a logic for shared mutable data structures, in: LICS'02, IEEE, 2002, pp. 55–74.
[6] D. Pym, J. Spring, P. O'Hearn, Why separation logic works, Philosophy & Technology 32 (3) (2019) 483–516, https://doi.org/10.1007/s13347-018-0312-8.
[7] C. Areces, B. ten Cate, Hybrid logics, in: P. Blackburn, F. Wolter, J. van Benthem (Eds.), Handbook of Modal Logic, Elsevier, 2007, pp. 821–868.
[8] J. van Benthem, An essay on sabotage and obstruction, in: Mechanizing Mathematical Reasoning, 2005, pp. 268–276.
[9] C. Löding, P. Rohde, Model checking and satisfiability for sabotage modal logic, in: P. Pandya, J. Radhakrishnan (Eds.), FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science: 23rd Conference, Proceedings, Mumbai, India, December 15–17, 2003, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 302–313.
[10] C. Areces, R. Fervari, G. Hoffmann, Relation-changing modal operators, Logic Journal of the IGPL 23 (4) (2015) 601–627, https://doi.org/10.1093/jigpal/jzv020.
[11] D. Gabbay, Introducing reactive Kripke semantics and arc accessibility, in: A. Avron, N. Dershowitz, A. Rabinovich (Eds.), Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, in: Lecture Notes in Computer Science, vol. 4800, Springer, 2008, pp. 292–341.
[12] D. Gabbay, Reactive Kripke Semantics, Cognitive Technologies, Springer, 2013.
[13] G. Aucher, P. Balbiani, L.F. del Cerro, A. Herzig, Global and local graph modifiers, ENTCS 231 (2009) 293–307.
[14] C. Lutz, Complexity and succinctness of public announcement logic, in: Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '06, New York, NY, USA, 2006, pp. 137–143.
[15] J. Plaza, Logics of public communications, Synthese 158 (2) (2007) 165–179, first published in 1989.
[16] H. van Ditmarsch, W. van der Hoek, B. Kooi, Dynamic epistemic logic with assignment, in: AAMAS 2005, ACM, 2005, pp. 141–148.
[17] C. Areces, D. Figueira, S. Figueira, S. Mera, Expressive power and decidability for memory logics, in: Logic, Language, Information and Computation, in: Lecture Notes in Computer Science, vol. 5110, Springer, 2008, pp. 56–68.
[18] S. Mera, Modal memory logics, Ph.D. thesis, Université Henri Poincaré, Nancy, France and Universidad de Buenos Aires, Argentina, 2009.
[19] C. Areces, D. Figueira, S. Figueira, S. Mera, The expressive power of memory logics, The Review of Symbolic Logic 4 (2) (2011) 290–318.
[20] C. Areces, P. Blackburn, M. Marx, Hybrid logics: characterization, interpolation and complexity, Journal of Symbolic Logic 66 (3) (2001) 977–1010.
[21] D. Grossi, S. Rey, Credulous acceptability, poison games and modal logic, in: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13–17, 2019, 2019, pp. 1994–1996.
[22] G. Aucher, J. van Benthem, D. Grossi, Modal logics of sabotage revisited, Journal of Logic and Computation 28 (2) (2018) 269–303, https://doi.org/10.1093/logcom/exx034.
[23] C. Areces, R. Fervari, G. Hoffmann, Moving arrows and four model checking results, in: WoLLIC 2012, in: LNCS, vol. 7456, Springer, 2012, pp. 142–153.
[24] C. Areces, R. Fervari, G. Hoffmann, Swap logic, Logic Journal of the IGPL 22 (2) (2014) 309–332, https://doi.org/10.1093/jigpal/jzt030.
[25] R. Fervari, Relation-changing modal logics, Ph.D. thesis, Universidad Nacional de Córdoba, Argentina, 2014.
[26] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development: Coq'Art the Calculus of Inductive Constructions, 1st edition, Springer Publishing Company, Incorporated, 2010.
[27] W.A. Howard, The formulae-as-types notion of construction, To HB Curry: essays on combinatory logic, lambda calculus and formalism 44 (1980) 479–490.
[28] G. Gonthier, Formal proof–the four-color theorem, Notices of the AMS 55 (11) (2008) 1382–1393.
[29] T.C. Hales, M. Adams, G. Bauer, D.T. Dang, J. Harrison, T.L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T.T. Nguyen, T.Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J.M. Rute, A. Solovyev, A.H.T. Ta, T.N. Tran, D.T. Trieu, J. Urban, K.K. Vu, R. Zumkeller, A formal proof of the Kepler conjecture, Forum of Mathematics, Pi 5 (2017) e2.
[30] K. Slind, M. Norrish, A brief overview of hol4, in: International Conference on Theorem Proving in Higher Order Logics, Springer, 2008, pp. 28–32.
[31] L.C. Paulson, Isabelle: A Generic Theorem Prover, Vol. 828, Springer Science & Business Media, 1994.
[32] P. de Wind, Modal Logic in Coq, Vrije Universiteit, Amsterdam, 2001.
[33] F. Wolter, M. Zakharyaschev, Intuitionistic Modal Logic, Springer, Netherlands, 1999, pp. 227–238.
[34] G. Priest, An Introduction to Non-classical Logic: From If to Is, 2nd edition, Cambridge U Press, 2000.

[35] C. Benzmüller, M. Claus, N. Sultana, Systematic verification of the modal logic cube in isabelle/hol, in: Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015, 2015, pp. 27–41.
[36] C. D'Abrera, R. Goré, Verified synthesis of (very simple) Sahlqvist correspondents via coq, in: AiML 2018, Short Presentations, College Publications, 2018, pp. 26–30.
[37] M. Wu, R. Goré, Verified decision procedures for modal logics, in: ITP 2019, in: LIPIcs, vol. 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 31:1–31:19.
[38] L. de Moura, S. Kong, J. Avigad, F. van Doorn, J. von Raumer, The lean theorem prover (system description), in: CADE 2015, 2015, pp. 378–388.
[39] L. González-Huesca, F.E. Miranda-Perea, P.S. Linares-Arévalo, Axiomatic and dual systems for constructive necessity, a formally verified equivalence, Journal of Applied Non-Classical Logics 29 (3) (2019) 255–287, https://doi.org/10.1080/11663081.2019.1647653.
[40] B. Xavier, C. Olarte, G. Reis, V. Nigam, Mechanizing focused linear logic in coq, ENTCS 338 (2018) 219–236.
[41] B. Bohrer, A. Platzer, Toward structured proofs for dynamic logics, CoRR, arXiv:1908.05535 [abs], 2019.
[42] S. Mitsch, A. Platzer, The keymaera X proof IDE - concepts on usability in hybrid systems theorem proving, in: F-IDE@FM 2016, in: EPTCS, vol. 240, 2016, pp. 67–81.
[43] R. Fervari, F. Trucco, B. Ziliani, Mechanizing bisimulation theorems for relation-changing logics in Coq, in: DaLí 2019, in: LNCS, vol. 12005, Springer, 2019, pp. 3–18.
[44] J.-O. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas, D. Dreyer, Mtac2: typed tactics for backward reasoning in Coq, Proc. ACM Program. Lang. 2 (ICFP) (2018) 78:1–78:31, https://doi.org/10.1145/3236773.
[45] A. Mahboubi, E. Tassi, Mathematical components, https://math-comp.github.io/mcb/, 2018.
[46] J. van Benthem, Modal correspondence theory, in: Handbook of Philosophical Logic, vol. 2, 1984, pp. 167–247.
[47] P. Rohde, On games and logics over dynamically changing structures, Ph.D. thesis, RWTH Aachen, 2006.
[48] B. Kooi, B. Renne, Generalized arrow update logic, in: Theoretical Aspects of Rationality and Knowledge, in: Proceedings of the Thirteenth Conference, 2011, pp. 205–211.
[49] B. Kooi, B. Renne, Arrow update logic, Review of Symbolic Logic 4 (4) (2011) 536–559.
[50] P. Girard, J. Seligman, F. Liu, General dynamic dynamic logic, in: AiML 2012, College Publications, 2012, pp. 239–260.
[51] J. van Benthem, F. Liu, Dynamic logic of preference upgrade, Journal of Applied Non-Classical Logics 17 (2) (2007) 157–182, https://doi.org/10.3166/jancl.17.157-182.
[52] R. Fervari, F.R. Velázquez-Quesada, Dynamic epistemic logics of introspection, in: DaLí 2017, in: LNCS, vol. 10669, Springer, 2017, pp. 82–97.
[53] R. Fervari, F.R. Velázquez-Quesada, Introspection as an action in relational models, Journal of Logical and Algebraic Methods in Programming 108 (2019) 1–23.
[54] D. Harel, Dynamic Logic, Foundations of Computing, The MIT Press, 2000.
[55] J. van Benthem, Model correspondence theory, Ph.D. thesis, University of Amsterdam, 1976.
[56] S. Demri, M. Deters, Separation logics and modalities: a survey, Journal of Applied Non-Classical Logics 25 (1) (2015) 50–99, https://doi.org/10.1080/11663081.2015.1018801.
[57] S. Demri, R. Fervari, On the complexity of modal separation logics, in: AiML 2018, College Publications, 2018, pp. 179–198.
[58] S. Demri, R. Fervari, A. Mansutti, Axiomatising logics with separating conjunction and modalities, in: JELIA 2019, in: LNCS, vol. 11468, Springer, 2019, pp. 692–708.
[59] S. Demri, R. Fervari, The power of modal separation logics, Journal of Logic and Computation 29 (8) (2019) 1139–1184, https://doi.org/10.1093/logcom/exz019.