



Lean Formalization of Completeness Proof for Coalition Logic with Common Knowledge

Kai Obendrauf  

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Anne Baanen   

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Patrick Koopmann   

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Vera Stebletsova  

Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands

Abstract

Coalition Logic (CL) is a well-known formalism for reasoning about the strategic abilities of groups of agents in multi-agent systems. Coalition Logic with Common Knowledge (CLC) extends CL with operators from epistemic logics, and thus with the ability to model the individual and common knowledge of agents. We have formalized the syntax and semantics of both logics in the interactive theorem prover Lean 4, and used it to prove soundness and completeness of its axiomatization. Our formalization uses the type class system to generalize over different aspects of CLC, thus allowing us to reuse some of to prove properties in related logics such as CL and CLK (CL with individual knowledge).

2012 ACM Subject Classification Security and privacy → Logic and verification; Mathematics of computing → Mathematical software; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Multi-agent systems, Coalition Logic, Epistemic Logic, common knowledge, completeness, formal methods, Lean prover

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.28

Related Version *Extended Version*: <https://zenodo.org/records/12582709>

Supplementary Material

Software (Source Code): <https://github.com/kaiobendrauf/cl-lean> [20]
archived at [swh:1:dir:5679444e6dc3b26cd7f1c54786bff9be89541c19](https://sw.haskell.org/5679444e6dc3b26cd7f1c54786bff9be89541c19)

Funding *Anne Baanen*: NWO Vidi grant No. 016.Vidi.189.037, Lean Forward.

1 Introduction

Computers rarely work in isolation, rather they constantly interact with both human users and other devices. Such interconnected systems can range from household Internet of Things (IoT) devices, working towards creating a useful digital home for a user [2], to safety-critical systems for metros that need to account for multiple trains [15]. Correctly designing and verifying such systems is an important goal of research in Artificial Intelligence, specifically in the field of Multi Agent Systems (MAS) [11, 13, 27]. The large number of agents and simultaneous goals involved in these interactions make them highly complex. Furthermore, computers in such systems must often operate with imperfect information [26], for instance because they have limited input about the external environment [13]. It can therefore be difficult to maintain an overview of whether a system has been correctly programmed to always meet its requirements, highlighting the need for formal modelling and programmatic verification [27].



© Kai Obendrauf, Anne Baanen, Patrick Koopmann, and Vera Stebletsova;
licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 28; pp. 28:1–28:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we focus on Coalition Logic with Common Knowledge (CLC) to model such systems and their requirements. Coalition Logic (CL) was introduced by Marc Pauly in 2002 [22] for reasoning about abilities of agents, and is a popular logic in MAS research [1]. CL introduces an effectivity operator, which describes whether some group of agents is effective for ensuring some outcome, regardless of the actions of other agents. CL was later extended by Ågotnes and Alechina [1] into CLC by adding operators from epistemic logic for individual and common knowledge.

The current paper aims to build a foundation for CLC formalizations for MAS by investigating how CLC can be defined and reasoned over using the Lean prover [8]. Lean is an interactive proof assistant based on dependent type theory, and its mathematical library Mathlib [25] is a rapidly growing community-driven project that we made thankful use of. In this work, we use Lean to formalize the syntax and semantics of CLC and formalize the soundness and completeness theorem together with the finite model property of CLC as given by Ågotnes and Alechina [1]. Formalizing these proofs allows us to check that the syntax and semantics of CLC defined in Lean relate to one another as expected. Additionally, doing so demonstrates that these definitions can be used in nontrivial proofs about CLC.

Since we closely follow the Ågotnes and Alechina proof in our formalization, we will focus on the larger scale proof engineering aspects and show only relevant excerpts of these proofs. A full, **sorry**-free, version of our code is available online at <https://github.com/kaiobendrauf/c1-lean>. The formalization of CLC is part of a larger work [21], where the entire proofs and their formalization are given in as much detail as possible. This larger work [21] also formalizes soundness and completeness of CL. Although the current paper focuses on CLC, we give special attention to lemmas and definitions that are also used in the completeness proof for CL. Thus we illustrate the ways in which we prioritize generalizability and reusability in our Lean implementation. The intention of this design choice is to make our formalization easier for future work to extend.

In the following sections we give a brief overview of existing work formalizing logics related to CLC in Section 2 and an overview of the Lean prover and its mathematical library in Section 3. We give a detailed description of the syntax, semantics and axiomatic system of CLC in Section 4. We describe our definition of CLC in Lean in Section 5. Our formalization of the soundness of CLC is described in some detail in Section 6. Section 7 notes a method of making our formalization reusable for other logics. Finally, using these definitions we will prove in Lean that CLC is complete in Section 8.

2 Related work

The formalization of modal logics in proof assistants is an active area of research. To our knowledge, CLC has not yet been formalized in any proof assistant, however our work builds on related work on formalizing Epistemic Logics (EL) and CL. We start by describing work on CL. Nalon et al. [18] present a prototype automated reasoning tool for CL, based on a sound, complete, and terminating resolution-based calculus for CL in SWI-Prolog. On the other hand, Baston and Capretta [5] propose how to formalize the relation between strategic games and the effectivity operator in CL. These works provide support that CL can be defined and reasoned with in proof assistants. However, to the best of our knowledge, at this point in time there are no current works that formalize completeness of CL in any proof assistant, nor has any project formalized CL in Lean.

In contrast there are several existing formalizations of EL both in Lean [6, 17, 19] and other proof assistants [7, 9, 16]. In Lean, the first of these is the completeness proof of EL (S5) by Bentzen [6]. Following this, both Neeley [19] and Li [17] formalized completeness

again, but with different approaches, showing how flexibly such proofs can be implemented in Lean. We will, when possible, defer to these existing works on formalizing EL in Lean for guidance on implementation choices. Most often, we use ideas by Neeley [19] as she uses the same type of proof as Ågotnes and Alechina [1] while being particularly detailed about her design decisions. Furthermore, this work formalizes several logics, thus we know the design decisions generalize to multiple types of logic.

3 The Lean prover and Mathlib

We used the Lean theorem prover [8] in our formalization. Lean is an interactive theorem prover based on the Calculus of Inductive Constructions, featuring proof irrelevance, quotient types and classical reasoning. These features are used ubiquitously throughout the flagship mathematical library for Lean, Mathlib [25], which we used as a starting point for our own formalization. An introduction to Lean can be found at [3].

A characteristic aspect of Mathlib is its use of *typeclasses* to organize mathematical theories. Lean’s typeclass system extends the class mechanism introduced for operator overloading in Haskell [28], and are used to associate types with both operators and axioms about these operators. Moreover, the typeclass system permits extending structures, so that, for example, any theorem declared for a `Monoid M` will automatically apply to a type `G` for which a `Group G` instance exists. The typeclass system is invoked by placing parameters to declarations between square brackets. An *instance synthesis* algorithm is used to supply values for these parameters automatically, through a variation on depth-first search [23].

In 2023, Mathlib was ported from Lean 3 to the newly released Lean 4, a port that required substantial changes in notation and design choices. Our project was originally written for Lean 3 and after the proofs were completed, we ported it to Lean 4. The code we present in our paper is an abridged version of the Lean 4-compatible source code, using Lean version 4.4.0-rc1 and Mathlib commit 98fe17fd. Although this paper omits many proof steps for presentational purposes, in our accompanying formalization all proofs are complete and `sorry`-free.

4 Coalition Logic with Common Knowledge

We recall the syntax and semantics of CLC, as well as the axiomatization, following Ågotnes and Alechina [1], in their work extending CL [22].

Based on a finite, non-empty set N of *agents*, and a set Φ_0 of *atomic propositions*, CLC formulas are constructed using the usual propositional logic operators, the CL *effectivity* operator $[G]$, where $G \subseteq N$, and two epistemic operators: K_i for individual knowledge, where $i \in N$, and C_G for common knowledge. Formally, CLC formulas are defined by the following BNF grammar:

$$\varphi := \perp \mid p \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid [G]\varphi \mid K_i\varphi \mid C_G\varphi$$

where $p \in \Phi_0$, $G \subseteq N$ and $i \in N$. We note that our syntax here is slightly different from that of Ågotnes and Alechina [1], as we allow the case when $G = \emptyset$ for the $C_G\varphi$ operator. Additionally, based on Neeley [19], we use a non-minimal set of propositional operators as this simplifies our proofs in Lean.

$[G]\varphi$ expresses the *effectivity* of a coalition to achieve φ . Intuitively, $[G]\varphi$ can be read as “coalition group G can ensure φ , regardless of the actions of agents not in the coalition”. K_i expresses the knowledge of an agent $i \in N$. Thus, $K_i\varphi$ can be intuitively read as “agent i

knows φ ". This individual knowledge can be extended to groups via the derived operator E_G , using the conjunction of individual knowledge. Specifically, for $G \subseteq N$, the notation $E_G\varphi$ is defined as $E_G\varphi := \bigwedge_{i \in G} K_i\varphi$ and reads as "everyone in group G knows φ ". $C_G\varphi$ expresses that group G has common knowledge of φ . Intuitively this can be read as "everyone in group G knows φ , and they all know that they all know φ , and they all know that they all know that they all know φ and so on".

The semantics of CLC is based on *epistemic coalition* frames and models. An epistemic coalition model contains an *epistemic accessibility relation* \sim_i for each agent $i \in N$. These are equivalence relations that model what each agent knows. Specifically, if $(s, t) \in \sim_i$ for some agent i , written as $s \sim_i t$, then agent i cannot differentiate state s and state t .

Additionally epistemic coalition frames and models contain an *effectivity structure* E which represents the effectivity of coalitions. Given a non-empty set S of states, E maps a state and subset of N to a set of subsets of S , i.e. $E : S \rightarrow \mathcal{P}(N) \rightarrow \mathcal{P}(\mathcal{P}(S))$. Note that, given some state $s \in S$ and set of agents $G \subseteq N$, $E(s)(G)$ denotes a set of sets of states. Intuitively, if $X \in E(s)(G)$, the coalition G must have some joint strategy in state s such that, no matter the strategy of agents not in the coalition, we are guaranteed to end up in some $t \in X$. In this way the effectivity structure models the ability of coalitions to ensure some (sets of) outcomes while abstracting away specific actions and strategies. In order adequately model a coalition's effectivity we require specific properties to hold, which are collectively defined by the concept of *true playability*.

► **Definition 1** (True Playability). *A truly playable effectivity structure is an effectivity structure E such that for any state s , $E(s)$ meets the following 6 conditions [1, Section 2.1].*

1. $E(s)$ is live: for every $G \subseteq N$, $\emptyset \notin E(s)(G)$
2. $E(s)$ is safe: for every $G \subseteq N$, $S \in E(s)(G)$
3. $E(s)$ is N -maximal: for every $X \subseteq S$, if $(S \setminus X) \notin E(s)(\emptyset)$, then $X \in E(s)(N)$
4. $E(s)$ is outcome monotonic: for every $G \subseteq N$ and $X, Y \subseteq S$, if $X \in E(s)(G)$ and $X \subseteq Y$, then also $Y \in E(s)(G)$
5. $E(s)$ is superadditive: for all $C, D \subseteq N$ where $C \cap D = \emptyset$, and all $X, Y \subseteq S$, if $X \in E(s)(C)$ and $Y \in E(s)(D)$, then $X \cap Y \in E(s)(C \cup D)$
6. $E(s)(\emptyset)$ is principal: there exists an $X \in E(s)(\emptyset)$ such that for every $Y \in E(s)(\emptyset)$, we have $X \subseteq Y$.

We have now everything to define epistemic coalition frames and models formally.

► **Definition 2.** *An epistemic coalition frame is a tuple $F = (S, E, \{\sim_i : i \in N\})$, where*

- S is a non-empty set of states,
- $E : S \rightarrow (\mathcal{P}(N) \rightarrow \mathcal{P}(\mathcal{P}(S)))$ is a truly playable effectivity structure, and
- $\sim_i \subseteq S \times S$ is an equivalence relation, the epistemic accessibility relation over S for agent i .

► **Definition 3.** *An epistemic coalition model is a tuple $M = (F, V)$, where:*

- F is an epistemic coalition frame, and
- $V : \Phi_0 \rightarrow \mathcal{P}(S)$ is the usual valuation function, assigning to each $p \in \Phi_0$ some set of states $V(p) \subseteq S$.

Based on an epistemic coalition model $M = (F, V)$, where $F = (S, E, \{\sim_i : i \in N\})$, and some state $s \in S$, we can now define what it means for a CLC formula ϕ to be true in s (written as $M, s \models \phi$). Truth of $[G]\varphi$ relates to the effectivity structure: if in state s group G is effective in bringing about φ , then G must be able to restrict the possible next states to some set containing only states where φ is true. $K_i\varphi$ relates intuitively to the \sim_i relation: if

■ **Table 1** Axiomatization of CLC.

(Prop)	Prop. tautologies	(K)	$\vdash K_i(\varphi \rightarrow \psi) \rightarrow (K_i\varphi \rightarrow K_i\psi)$	(T)	$\vdash K_i\varphi \rightarrow \varphi$
(4)	$\vdash K_i\varphi \rightarrow K_iK_i\varphi$	(5)	$\vdash \neg K_i\varphi \rightarrow K_i\neg K_i\varphi$	(C)	$\vdash C_G\varphi \rightarrow E_G(\varphi \wedge C_G\varphi)$
(\perp)	$\vdash \neg[G]\perp$	(\top)	$\vdash [G]\top$	(N)	$\vdash (\neg[\emptyset]\neg\varphi) \rightarrow [N]\varphi$
(S)	$\vdash ([G]\varphi \wedge [F]\psi) \rightarrow [G \cup F](\varphi \wedge \psi), \text{ if } G \cap F = \emptyset$				
(MP)	$\vdash \varphi, \varphi \rightarrow \psi \Rightarrow \vdash \psi$	(RN)	$\vdash \varphi \Rightarrow \vdash K_i\varphi$	(Eq)	$\vdash \varphi \leftrightarrow \psi \Rightarrow \vdash [G]\varphi \leftrightarrow [G]\psi$
(RC)	$\vdash \psi \rightarrow E_G(\varphi \wedge \psi) \Rightarrow \vdash \psi \rightarrow C_G\varphi$				

agent i knows φ in state s , then φ must be true in all states that i cannot distinguish from s . The operator $C_G\varphi$ is a little more complex, as here we need to consider paths through epistemic relations. For readability, we write $(s, t) \in (\bigcup_{i \in G} \sim_i)^*$ as $s \approx_G t$. If group G has common knowledge of φ in state s , then φ must be true in all states t such that $s \approx_G t$. Truth in M, s now defined as follows.

$M, s \not\models \perp$	
$M, s \models p$	iff $p \in \Phi_0$ and $s \in V(p)$
$M, s \models \varphi \wedge \psi$	iff $M, s \models \varphi$ and $M, s \models \psi$
$M, s \models \varphi \rightarrow \psi$	iff $M, s \models \varphi \Rightarrow M, s \models \psi$
$M, s \models [G]\varphi$	iff $\{s \in S \mid M, s \models \varphi\} \in E(s)(G)$
$M, s \models K_i\varphi$	iff $\forall t \in S, s \sim_i t \Rightarrow M, t \models \varphi$
$M, s \models C_G\varphi$	iff $\forall t \in S, s \approx_G t \Rightarrow M, t \models \varphi$

As usual, φ is *valid* in a model ($M \models \varphi$) if it is true in every state of the model and is *globally valid* ($\models \varphi$) if it is valid in all models.

The axiomatization of CLC can be seen in Table 1.

5 Formalizing the Syntax and Semantics in Lean

To formalize the syntax of CLC in Lean, we use a deep embedding, which allows us to prove metatheoretical results about the logic such as soundness and completeness [14, 19]. Thus, in Lean, the language of CLC formulas is defined as an inductive type, meaning the smallest type closed under the operators `bot`, `var`, `and`, `imp`, `eff`, `K` and `C`.

```

inductive formCLC (agents : Type) : Type
| bot
  : formCLC agents
| var (n : Nat)
  : formCLC agents
| and (φ ψ : formCLC agents)
  : formCLC agents
| imp (φ ψ : formCLC agents)
  : formCLC agents
| eff (G : Set agents) (φ : formCLC agents)
  : formCLC agents
| K (a : agents) (φ : formCLC agents)
  : formCLC agents
| C (G : Set agents) (φ : formCLC agents)
  : formCLC agents

```

The inductive type is parameterized over an arbitrary type `agents`. At this point we do not require that only finitely many agents appear in the formula. Instead, we will apply this assumption only to those theorems whose proofs require it, guided by the automated proof checking done by Lean.

28:6 Formalizing Completeness of CLC in Lean

To define the semantics, we first define effectivity structures:

```
def effectivity_struct (agents states : Type) :=
  states → Set agents → Set (Set states)
```

To represent a playable effectivity structure, we create a 6-tuple to link the effectivity function itself to the 5 playability requirements. Thus, we need to store tuples of a certain shape, which in Lean we do using a `structure` data type:

```
structure truly_playable_effectivity_struct (agents states : Type) :=
  (E      : effectivity_struct agents states)
  (liveness : ∀ s : states, ∀ G : Set agents, ∅ ∉ E s G)
  (safety  : ∀ s : states, ∀ G : Set agents, univ ∈ E s G)
  (N_max   : ∀ s : states, ∀ X : Set states, Xc ∉ E s ∅ → X ∈ E s univ)
  (mono    : ∀ s : states, ∀ G : Set agents, ∀ X Y : Set states,
             X ⊆ Y → X ∈ E s G → Y ∈ E s G)
  (superadd : ∀ s : states, ∀ G F : Set agents, ∀ X Y : Set states,
             X ∈ E s G → Y ∈ E s F → G ∩ F = ∅ →
             X ∩ Y ∈ E s (G ∪ F))
  (principal_E_s_empty : ∀ s, ∃ X, X ∈ E s ∅ ∧ ∀ Y, Y ∈ E s ∅ → X ⊆ Y)
```

Comparing the semantics defined in Section 4, we can see a particular difference in the treatment of sets: where informally we write $X \in E(s)(N)$, Lean writes $X \in E\ s\ univ$. Since Lean is based on type theory, it distinguishes `agents : Type` from its universal set `univ : Set agents`. Apart from this distinction, the conditions translate straightforwardly.

Epistemic coalition frames and models are then defined as follows:

```
structure frameECL (agents : Type) :=
  (states      : Type)
  (hs         : Nonempty states)
  (E          : truly_playable_effectivity_struct agents states)
  (rel        : agents → states → Set states)
  (rfl       : ∀ i s, s ∈ rel i s)
  (sym       : ∀ i s t, t ∈ rel i s → s ∈ rel i t)
  (trans     : ∀ i s t u, t ∈ rel i s → u ∈ rel i t → u ∈ rel i s)
```

```
structure modelECL (agents : Type) :=
  (f : frameECL agents)
  (v : ℕ → Set f.states)
```

To encode semantic entailment, we first formalize the *common knowledge path* recursively defined predicate that we call a `C_path`.

```
inductive C_path {agents : Type} {m : modelECL agents} (G : Set agents) :
  m.f.states → m.f.states → Prop
| done (hi : i ∈ G) (hst : t ∈ m.f.rel i s) : C_path G s t
| next (hi : i ∈ G) (hsu : u ∈ m.f.rel i s) (ih : C_path G u t) :
  C_path G s t
```

Intuitively, we say given a coalition G that there is a path from state s to state t if we can give, for some $n \geq 1$, some list i_0, i_1, \dots, i_n of agents in G , as well as some list u_1, u_2, \dots, u_n of states, such that $s \sim_{i_0} u_1, u_1 \sim_{i_1} u_2, \dots, u_n \sim_{i_n} t$. $s \approx_G t$ then means that there is a

C_path from s to t , where every agent in the list of agents is also in G . From here, defining semantic entailment is straightforward, so we show only the non-propositional cases:

```
def s_entails_CLC {agents : Type} (m : modeleCL agents) (s : m.f.states) :
  formCLC agents → Prop
...
| (.[G] φ)   => {t : m.f.states | s_entails_CLC m t φ} ∈ m.f.E.E s G
| (.K i φ)   => ∀ t : m.f.states, t ∈ m.f.rel i s → s_entails_CLC m t φ
| (.C G φ)   => ∀ t : m.f.states, C_path G s t → s_entails_CLC m t φ
```

6 Formalizing Soundness

The axiomatization of CLC (Table 1) is defined as an inductive predicate, that is, as an inductively defined proposition [4]. An inductive predicate is defined as the smallest predicate closed under a set of proof steps. Thus, an inductive predicate contains all proofs constructed from a finite tree of proof steps. This mirrors how the set of formulas provable in an axiomatic system is the smallest set closed under rule applications. The translation to Lean is thus very straightforward and omitted here. Before we come to the more challenging proof of completeness of this system, we prove its soundness.

► **Theorem 4** (Soundness of CLC [1, Lemma 1]). $\forall \varphi, \vdash \varphi \Rightarrow \models \varphi$

Despite the proof itself being simple, translating it into Lean is not entirely straightforward. We prove this theorem by structural induction on the proof of $\vdash \varphi$. Most of the cases can be proven directly from the given axiom. Note that axioms (\perp), (\top), (**N**), (**M**) and (**S**) relate directly to the first five true playability requirements, and axioms (**T**), (**4**) and (**5**) relate to the fact that epistemic relations are equivalence relations.

The cases (**C**) and (**RC**) are a little more complex, as they involve the C_G operator. To show that a formula of the form $C_G\varphi$ is true, we need to reason about the common knowledge relation \approx_G . More specifically, in Lean we have to look for C_paths between states. To illustrate how this is done, we look closer at the case for Axiom (**C**). Given $M, s \models C_G\varphi$, we need to show $M, s \models E_G(\varphi \wedge C_G\varphi)$, which gives the following goal after simplifying:

```
h : M, s ⊨ C G φ
hi : i ∈ G
hts : t ∈ M.f.rel i s
⊢ M, t ⊨ φ ∧ C G φ
```

For the first half of the conjunction, we apply the hypothesis h , so it remains to prove that $C_path\ G\ s\ t$ holds. In this case the path will have length one, so that the constructor $C_path.done$ applies, and our existing hypothesis $hts : t \in M.f.rel\ i\ s$ concludes this case.

For the second half of the conjunction, we get the following goal after simplification:

```
h : M, s ⊨ C G φ
hi : i ∈ G
hts : t ∈ M.f.rel i s
htu : C_path G t u
⊢ M, u ⊨ φ
```

Intuitively for any state u such that $t \approx_G u$, we must show $M, u \models \varphi$. Again we apply h , leaving us to show $C_path\ G\ s\ u$ which must hold when we extend $C_path\ G\ t\ u$ by first using agent i to pass from s to t . This corresponds to the $C_path.next$ constructor of C_path , using the hypotheses hts and htu to discharge the remaining goals.

7 Creating reusable definitions in Lean

Before tackling the completeness proof for CLC, we note that the proof relies in large part on lemmas and definitions taken from Pauly’s completeness for CL [22]. In paper proofs such reuse is trivial, but in Lean lemmas and definitions only apply to the syntax they are defined on, since the syntax and proof system for each logic form a distinct inductive type. Our formalization therefore gives special attention to reusability using the *typeclass* system of Lean, to limit the need for redundant copies of code for each logic. Specifically, we make use of the fact that one logic commonly extends another by adding new operators and axioms. In Lean we define a `class` for some logic in such a way that all extensions of that logic are an `instance` of that `class`. We can then construct definitions and proofs in Lean that apply to any logic that is an instance of that `class`. Doing so allows our Lean results to be reused across different logics.

We start by creating a typeclass for logics whose syntax extends that of propositional logic. More precisely, an instance of `Pformula form`

```
class Pformula (form : Type) :=
  (bot : form)
  (var : ℕ → form)
  (and : form → form → form)
  (imp : form → form → form)
```

We also introduce notation for formulas: \perp , \wedge , \rightarrow , \top , \neg , \vee , \iff . Next, we demonstrate that the language of CLC formulas `formCLC` extends propositional logic by registering an `instance`.

```
instance formulaCLC {agents : Type} : Pformula (formCLC agents) :=
  { bot := formCLC.bot,
    var := formCLC.var,
    and := formCLC.and,
    imp := formCLC.imp, }
```

We can then make our formula constructions generic over all syntaxes that have a `Pformula` instance, and Lean will automatically infer this instance when applying these constructions. For instance, the following definition gives the conjunction of a finite list of formulas.

```
def finite_conjunction {form : Type} [Pformula form] : List form → form
| []      := ⊤
| (f :: fs) := f ∧ finite_conjunction fs
```

Since all provable propositional formulas are also provable in logics that extend propositional logic, we also introduce a `class Pformula_ax (form : Type) [Pformula form]` that denotes the existence of a provability predicate \vdash such that $\vdash \varphi$ holds for all formulas φ provable by the axioms of propositional logic.

We create three more typeclasses relevant to CLC. Logics (extending) CL are instances of `class CLformula (agents : outParam Type) (form : Type) [Pformula_ax form]`, which specifies the additional operator and axioms associated with CL. Note that this typeclass inherits from `Pformula_ax` as CL extends propositional logic. Similarly we introduce `class Kformula (agents : outParam Type) (form : Type) [Pformula_ax form]` representing logics that extend propositional logic with individual knowledge. Lastly we create a typeclass for logics with common knowledge, which must therefore also contain individual

knowledge. This typeclass must therefore inherit from both `Pformula_ax` and `Kformula`:
`Cformula (agents : outParam Type) [hN : Fintype agents] (form : Type)
 [pf : Pformula_ax form] [kf : Kformula agents form].`

8 Formalizing Completeness

We begin by sketching the completeness proof for CLC [1, Corollary 1], which is based on a canonical model construction. For each consistent formula, we create a finite model for which that formula is true at some state. We focus on finite models because the 6th true playability condition, that $E(s)(\emptyset)$ is principal, is always met in finite models [12]. Doing so simplifies the proof that the effectivity structure in these models is truly playable. In the process we demonstrate that CLC has the finite model property.

We create such a finite model by first creating a single infinite canonical coalition model where every consistent formula is true in some state. This canonical coalition model is defined analogously to an epistemic coalition model, but without epistemic relations, and where the effectivity structure only meets the first 5 true playability conditions. Then, given some consistent formula φ , we filter the canonical coalition model and add epistemic relations to form a finite epistemic coalition model for which φ is true in some state.

8.1 Formalizing the canonical coalition model

We start by building the canonical coalition model. We define $M^C := (F^C, V^C)$, where $F^C := (S^C, E^C)$ as follows:

- S^C is the set of all maximal CLC-consistent sets of formulas.
- E^C is the playable effectivity structure:
 - $X \in E^C(s)(N)$ iff $\forall \varphi, \tilde{\varphi} \subseteq X^c \rightarrow [\emptyset]\varphi \notin s$, where $\tilde{\varphi} := \{t \in S^C \mid \varphi \in t\}$
 - $X \in E^C(s)(G)$ iff $\exists \varphi, \tilde{\varphi} \subseteq X \wedge [G]\varphi \in s$, when $G \neq N$
- V^C is the usual valuation function : $s \in V^C(p)$ iff $p \in s$.

A playable effectivity structure must meet the first 5 true playability conditions. A set Σ of formulas is consistent iff there are no $\sigma_1, \dots, \sigma_n \in \Sigma$ such that $\vdash (\sigma_1 \wedge \dots \wedge \sigma_n) \rightarrow \perp$. The proof that M^C is indeed a coalition model is analogous to the proof by Pauly [22, Lemma 5.2] for CL. In Lean, we use our generic classes for propositional logic and CL to define a canonical coalition model for any logic that extends CL, so long as that logics axiomatic system is consistent (as required by the hypotheses `hnpr : $\neg \vdash (\perp : \text{form})$`):

```
def canonical_model_CL [Nonempty agents]
  [Pformula_ax form] [CLformula agents form]
  (hnpr :  $\neg \vdash (\perp : \text{form})$ ) : modelCL agents
```

Note that this definition includes the proof that the defined effectivity structure is playable.

8.2 Filtering the canonical model

Given some φ , we filter S^C into a finite set of states S^f . We will prove the properties of S^C transfer to S^f and shows it enjoys some additional properties essential to constructing a playable model. We achieve this by creating a finite closure $cl(\varphi)$, defined as the smallest set satisfying the following:

1. For any $\psi \in cl(\varphi)$, all subformulas of ψ are also contained in $cl(\varphi)$.
2. For any $\psi \in cl(\varphi)$, if ψ is not of the form $\neg\chi$, then $\neg\psi \in cl(\varphi)$.
 ($cl(\varphi)$ is thus closed under single negations.)

3. If $C_G\varphi \in cl(\varphi)$, then for all $i \in G$, $K_i C_G\varphi \in cl(\varphi)$.
4. If $[G]\varphi \in cl(\varphi)$, then $C_G[G]\varphi \in cl(\varphi)$.

This definition is adjusted slightly compared to the work by Ågotnes and Alechina [1], as we allow the formula $C_\emptyset\psi$, and thus do not need to consider the case $G = \emptyset$ separately. Additionally, we change the first requirement such that all subformulas of any $\psi \in cl(\varphi)$ are contained in the closure, rather than just subformulas of φ . This change is needed to prove the truth lemma, where we will perform induction on an arbitrary $\psi \in cl(\varphi)$. For Ågotnes and Alechina [1] this adjusted requirement is already met when $cl(\varphi)$ contains all subformulas of φ , because their syntax is defined from different base operators. The closure definition thus illustrates that small implementation choices early in the formalization process can have unintended effects later in the proof that may not be immediately obvious. Luckily, the interactive environment of a theorem prover made the consequences of this change clear, and made the necessary changes easy to implement and test.

The set $cl(\varphi)$ can be built recursively on the structure of φ , and this is also how we define it in Lean. For instance, for the case $cl(C_G\psi)$, the closure must include $cl(\psi) \cup \{C_G\psi, \neg(C_G\psi)\} \cup \{K_i C_G\psi : i \in G\} \cup \{\neg(K_i C_G\psi) : i \in G\}$. Note that the sets $\{K_i C_G[G]\psi : i \in G\}$ and $\{\neg(K_i C_G\psi) : i \in G\}$ are finite, because G is finite. In Lean we define the union of these two sets as follows:

```

noncomputable def cl_C {agents : Type} [Fintype agents] (G : Set agents)
  (φ : formCLC agents) : Finset (formCLC agents) :=
  Finset.image (fun i => K i (C G φ)) (toFinset G) ∪
  Finset.image (fun i => (¬ K i (C G φ))) (toFinset G)

```

In addition to defining a set, the above definition also guarantees that the set is finite, using the `Finset` datatype. We create this resulting `Finset` by first mapping the set of agents G from a `Set` to a `Finset`. Lean can infer this is possible, because N is finite, as indicated by the hypothesis `[Fintype agents]`. Then we can take the image of G as desired.

In Lean, we then need to prove that our closure $cl(\phi)$, defined recursively on the structure of the formula ϕ indeed meets the four requirements described above. To do so, we first define a subformula as an inductive proposition with cases for each operator. For instance we define two cases for the \wedge -operator: `and_left` $\{\varphi \psi\} : \text{subformula } \varphi (\varphi \wedge \psi)$ and `and_right` $\{\varphi \psi\} : \text{subformula } \psi (\varphi \wedge \psi)$. Additionally, we add two cases for the requirements that our sub-formula definition must be reflexive and transitive. Given this definition, we tackle the four proofs about the closure. Although in a paper proof all four requirements are trivially met by definition of the closure, in Lean this is only the case for the last two. The first two requirements both need inductive proofs on φ where for every case we iteratively consider all possible $\psi \in cl(\varphi)$. For instance if $\varphi = \chi_l \wedge \chi_r$, we consider the cases where $\psi = \chi_l \wedge \chi_r$, $\psi = \neg(\chi_l \wedge \chi_r)$, $\psi \in cl(\chi_l)$ and $\psi \in cl(\chi_r)$. These proofs are not difficult, but considering each case creates long and tedious proofs.

Now that we have defined the closure, given some φ , we can filter M^C through $cl(\varphi)$, to construct a finite model $M^f := (F^f, V^f)$, where $F^f := (S^f, E^f, \{\sim_i^f : i \in N\})$. We construct M^f as follows:

$$\begin{array}{ll}
S^f & := \{(s^f) \mid s \in S^C\}, & \text{where } s^f := (s \cap cl(\varphi)) \\
E^f & := \begin{array}{l} X \in E^f(s)(N) \\ X \in E^f(s)(G \subset N) \end{array} & \begin{array}{l} \text{iff } \exists t \in S^C, s^f = t^f \text{ and } \widetilde{\phi}_X \in E^C(t)(N) \\ \text{iff } \forall t \in S^C, s^f = t^f \Rightarrow \widetilde{\phi}_X \in E^C(t)(G) \end{array} \\
\sim_i^f & := (s^f) \sim_i^f (t^f) & \text{iff } \{\varphi \mid K_i \varphi \in s^f\} = \{\varphi \mid K_i \varphi \in t^f\} \\
V^f & := s \in V(p) & \text{iff } p \in s,
\end{array}$$

where $\phi_X := \bigvee_{s^f \in X} \phi^{s^f}$ is the disjunction of a set of filtered states, $\phi^{s^f} := \bigwedge_{\psi \in s^f} \psi$ is the

conjunction of the formulas in a filtered state, and $\tilde{\psi} := \{t \in S^C \mid \psi \in t\}$. Note that S^f is finite because $cl(\varphi)$ is, and that \sim_i^f is an equivalence relation by definition. For this model we will use the notation $s^f \approx_G^f t^f := (s^f, t^f) \in (\bigcup_{i \in G} \sim_i^f)^*$ for the common knowledge path.

These definitions can be translated quite directly into Lean, although it might not look so direct, due to again having to distinguish between sets and finite sets in Lean. Thus, to define S^f in Lean, we start with $cl(\varphi)$, as this is a `Finset`. We take the powerset of $cl(\varphi)$, which Lean knows must also be finite. This finite powerset is filtered with `Finset.filter` to include only those elements s^f for which there exists some $s \in S^C$ such that $s^f = s \cap cl(\varphi)$. In order to check $s^f = s \cap cl(\varphi)$, we need both to be of the same data type and therefore convert both to sets. Finally, we pair each state s^f with a proof that it is produced by the filter, using `Finset.attach`.

```
def S_f {agents form : Type} (m : modelCL agents) [SetLike m.f.states form]
  (cl : form → Finset (form)) (φ : form) : Type :=
  Finset.attach (Finset.filter
    (λ sf => ∃ s : m.f.states, {x | x ∈ cl φ ∧ x ∈ s} = {x | x ∈ sf}))
    (Finset.powerset (cl φ))
```

Note that we do impose strong requirements on the model in the definition of S^f , so long as the states contain a set of formulas, as enforced by the hypothesis `[SetLike m.f.states form]`. Doing so allows us to keep our definition simpler and more generic, by removing the need for hypotheses needed to create our canonical model (for instance that N is nonempty).

Next we define the subformulas ϕ_X and ϕ^{s^f} which are needed to define E^f :

```
variable {agents form : Type} [Pformula form]
  {m : modelCL agents} [SetLike m.f.states form]
  {cl : form → Finset (form)} {φ : form}

noncomputable def phi_s_f (sf : S_f m cl φ) : form :=
  finite_conjunction (Finset.toList (sf.1))

noncomputable def phi_X_list : List (S_f m cl φ) → List form
  | List.nil => List.nil
  | (sf :: ss) => ((phi_s_f sf) :: phi_X_list ss)

noncomputable def phi_X_finset (X : Finset (S_f m cl φ)) : form :=
  finite_disjunction (phi_X_list (Finset.toList X))

noncomputable def phi_X_set (X : Set (S_f m cl φ)) : form :=
  phi_X_finset (Finite.toFinset (Set.toFinite X))
```

Here the `variable` statement adds the hypotheses to each subsequent declaration. Defining ϕ^{s^f} in Lean (`phi_s_f`) is as simple as converting our finite set to a list (putting the elements in an arbitrary order) and then creating a conjunction from that list. We then define ϕ_X in several steps. First we define a function `phi_X_list` to map X to $\{\phi^{s^f} : s^f \in X\}$. Next, we define ϕ_X for finite sets, as we can convert that finite set to a list, map it to formulas with `phi_X_list`, and then return the disjunction of that mapped list. Lastly, for the `Set` datatype, we define ϕ_X by converting to a `Finset`, which we can do because $X \subseteq S^f$ is a set of a finite type.

Although it would suffice logically to work with a `List` of filtered states, we provide the definition `phi_X_set` in higher generality for two reasons. Firstly, this approach more closely matches the definitions in the paper proof. Secondly, the definition should intuitively

not depend on a choice of order on the states, so we make this independence explicit in the datatypes. Splitting this definition up into three parts may seem to add complexity. However, it allows us to define lemmas about each data type, thereby breaking proofs down into smaller steps. We can prove lemmas more easily for a list, which is ordered, finite, and allows induction. Then, it is easy to show that if some lemma holds for a list, it must work for a list created from a (finite) set. Keeping track of the converted datatypes and how they relate to one another within a single lemma is non-trivial (and not always possible) in Lean. Thus in Lean we often prove some result about ϕ_X across three lemmas, one for each datatype: `Set`, `Finset` and `List`.

Given our definition(s) for ϕ_X , it is straightforward to define E^f , and then our whole model M^f . We thus omit these Lean translations.

8.3 Playability of the filtered canonical model

We prove that M^f meets the requirements for being a CLC model. We have to show that for an arbitrary state s^f in the filtered model, $E^f(s^f)$ is truly playable [1, Proposition 1]. This proof relies on the fact that E^f is defined from E^c . We are therefore able to exploit the fact that the first five true playability conditions hold in E^c to prove that they must also hold in E^f . In Lean we really benefit from our generic typeclasses here, as our proofs that E^c meets those playability conditions are written to hold for any logic that extends CL. Recall that the final true playability condition must hold in E^f because M^f is finite [12].

To formalize this proof, we first expand the proof by Ågnotes and Alechina, into a proof with similar levels of detail to a Lean formalization. We aim for a level of detail such that each step in our extended paper proof translates roughly into one step in Lean, possibly with some reshaping. To illustrate this procedure and the level of detail required we present our extended paper proof for Condition 3 of true playability (Definition 1), which was the most interesting to formalize in Lean.

$E^f(s^f)$ is N -maximal (for every $X \subseteq S^f$, if $(S^f \setminus X) \notin E^f(s^f)(\emptyset)$, then $X \in E^f(s^f)(N)$) is shown by the following sequence of proof steps:

1. Pick some $X \subseteq S^f$ such that $X^c = (S^f \setminus X) \notin E^f(s^f)(\emptyset)$.
2. $\neg(X^c \in E^f(s^f)(\emptyset))$, from Step 1.
3. $\neg(\forall t \in S^c : s^f = t^f \Rightarrow \widetilde{\phi_{X^c}} \in E^c(t)(\emptyset))$, from Step 2 and by definition of E^f .
4. $\exists t \in S^c : s^f = t^f$ and $\widetilde{\phi_{X^c}} \notin E^c(t)(\emptyset)$, from Step 3.
5. $\vdash \phi_{X^c} \leftrightarrow \neg\phi_X$, because $\vdash \phi_{S^f}$ and $\forall s, t \in S^{c'}, s^f \neq t^f \Rightarrow \vdash \phi^{s^f} \rightarrow \neg\phi^{t^f}$.
6. $\exists t \in S^c, s^f = t^f$ and $\neg\widetilde{\phi_X} \notin E^c(t)(\emptyset)$, from Step 4 and 5.
7. $\exists t \in S^c, s^f = t^f$ and $(\widetilde{\phi_X})^c \notin E^c(t)(\emptyset)$, from Step 6, because all $s \in S^c$ are maximally consistent.
8. $\exists t \in S^c, s^f = t^f$, and $\widetilde{\phi_X} \in E^c(t)(N)$, provided $s = t$, from Step 7, because $E^c(t)$ is N -maximal: $(\widetilde{\phi_X})^c \notin E^c(t)(\emptyset) \Rightarrow \widetilde{\phi_X} \in E^c(t)(N)$
9. $X \in E^f(s^f)(N)$, from Step 8, by definition of E^f .

In this expanded proof the only step that is not straightforward to formalize in Lean is Step 5. We elaborate on the process of formalizing this step as it is a good illustration of working with our Lean definition(s) of ϕ_X . For space we do not expand on the proofs that $\vdash \phi_{S^f}$ and $\forall s, t \in S^{c'}, s^f \neq t^f \Rightarrow \vdash \phi^{s^f} \rightarrow \neg\phi^{t^f}$. In Lean these proofs are given in the lemmas `univ_disjunct_provability` and `unique_s_f` respectively. To show $\vdash \phi_{X^c} \leftrightarrow \neg\phi_X$, in the \Leftarrow direction we first use a lemma defined elsewhere called `phi_X_set_disjunct_of_disjuncts`, which proves that $\vdash (\neg\phi_X \rightarrow \phi_Y) \Leftrightarrow \vdash (\phi_{X \cup Y})$, to change the goal to $\vdash \phi_{X \cup X^c}$. This lemma is trivial on paper by definition of ϕ_X , but requires

unfolding the definitions and their respective datatype in Lean. Next we change the goal to $\vdash \phi_{S^f}$, with the lemma `union_compl_self`, because the union of a set and its complement is the universe. Lastly, we can use `univ_disjunct_provability` to prove this goal:

```
apply (phi_X_set_disjunct_of_disjuncts _ _).mpr
rw [union_compl_self X]
apply univ_disjunct_provability
```

For the \Rightarrow direction, we cannot so immediately apply the relevant lemma `unique_s_f`, as this lemma refers to single elements (filtered states) in our sets X and X^c . We will eventually use an inductive proof to consider a single element in X . In Lean we will therefore need to work with a `list` datatype and will need to unfold our definitions of ϕ_X accordingly. We create a lemma per data type: `phi_X_set_unique`, `phi_X_finset_unique` and `phi_X_list_unique`, which show the slightly generalized result that $\vdash \phi_X \rightarrow \neg \phi_Y$ holds for any disjoint sets $X, Y \subseteq S^f$. Our actual proof simply applies this first lemma:

```
apply phi_X_set_unique hcl (compl_inter_self X)
```

where `compl_inter_self` is a lemma proving that a set and its complement are disjoint, and `hcl` is a proof that our closure is closed under single negations. The need to pass this condition of our closure forward highlights how verification makes explicit exactly when and for which purpose specific hypotheses are used. In this case we will eventually pass this hypothesis to the lemma `unique_s_f`.

The interesting work is within `phi_X_list_unique`. We have converted X into `sfs : List (S_f M cl φ)`, where `S_f M cl φ` corresponds to S^f for the canonical model M and Y into `tfs : List (S_f M cl φ)`. The proof is first inductive on X . The case when X is empty is trivial because ϕ_{\emptyset} is defined as \perp in Lean. Thus we unfold the definitions `phi_X_list` and `finite_disjunction`, and then use `explosion`, which represents the propositional lemma $\forall \psi, \vdash \perp \rightarrow \psi$.

```
induction' sfs with sf sfs ihsfs generalizing tfs
· -- sfs = []
  simp only [phi_X_list, finite_disjunction, explosion]
```

So let `sfs` contain at least one element `sf` at the head of the list, and call the rest of the list `sfs'`. Then we can split $\vdash (\phi^{s^f} \vee \phi_{X'}) \rightarrow \neg \phi_Y$ into two cases, where the latter follows from the induction hypothesis `ihsfs`. Note that the `sorry` keyword in the snippet below indicates a proof omitted from the paper for presentation purposes; the omitted proof is included below and in the full formalization source code.

```
· -- sfs = sf :: sfs'
  simp only [phi_X_list, finite_disjunction]
  apply or_cases
  --  $\vdash \phi_{sf} \rightarrow \neg \phi_{tfs}$ 
  · sorry -- Proof included below
  --  $\vdash \phi_{sfs'} \rightarrow \neg \phi_{tfs}$ 
  · apply ihsfs
    apply List.disjoint_of_disjoint_cons_left hdis
```

Here the lemma `List.disjoint_of_disjoint_cons_left` shows that `sfs'` and `tfs` must be disjoint when `sfs` and `tfs` are disjoint (represented by `hdis` in Lean).

28:14 Formalizing Completeness of CLC in Lean

For the case $\vdash \phi^{s^f} \rightarrow \neg \phi_Y$, we perform induction on Y (\mathbf{tfs}). Again here the base case of an empty list holds by propositional logic, so assume \mathbf{tfs} contains at least one element \mathbf{tf} at the head of the list, and call the rest of the list \mathbf{tfs}' . We now look at the contrapositive of our goal: $\vdash (\phi^{t^f} \vee \bigvee_{u^f \in \mathbf{tfs}'} \phi^{u^f}) \rightarrow \neg \phi^{s^f}$. Again we have two cases. For the former we can apply `lemma unique_s_f` where we prove $s^f \neq t^f$ by contradiction, based on the disjointness of both lists. The latter case is solved with the (new) induction hypothesis (`ih \mathbf{tfs}`).

```

· induction' tfs with tf tfs ihtfs
  · simp only [phi_X_list, finite_disjunction]
    exact mp _ _ (p1 _ _) iden -- applying propositional lemmas
  · simp [finite_disjunction] at *
    -- contrapositive
    refine contrapos.mp (cut dne (or_cases ?_ ?_))
    -- ⊢ phi tf → ¬ phi sf
    · apply unique_s_f hcl
      by_contra h
      simp only [h] at hdis
    -- ⊢ phi tfs' → ¬ phi sf (proved with ihtfs and propositional lemmas)
    · rw [←contrapos]
      exact cut dne (ihtfs hdis.2.1 hdis.2.2)

```

8.4 Truth lemma

Next we show that in the filtered canonical model all formulas contained in a state are also true in that state. Recall that M^f is the model created when filtering M^C through $cl(\varphi)$.

► **Lemma 5** (CLC Truth Lemma [1, Theorem 1]). *For all $s \in S^C$ and $\psi \in cl(\varphi)$, we have $M^f, s^f \models \psi$ iff $\psi \in s^f$.*

This proof is by induction on ψ . For space reasons we include only the proof for $C_G\psi$: $M^f, s^f \models C_G\psi$ iff $C_G\psi \in s^f$, and specifically the \Leftarrow direction, as this was the most interesting to formalize. Given $C_G\psi \in s^f$, and some state t^f such that $s^f \approx_G^f t^f$, we need to show $M^f, t^f \models \psi$. This proof is inductive on the common knowledge path from s^f to t^f . Thus, the details of this proof depend on how exactly we defined the common knowledge path in Lean.

Let the length of a common knowledge path be the number of states in the path between our first state (s^f) and our last state (t^f). In this case we may describe a common knowledge path from s^f to t^f as $\langle s^f, \sim_{i_0}^f, u_1^f, \sim_{i_1}^f, u_2^f, \dots, u_n^f, \sim_{i_n}^f, t^f \rangle$, such that $s_0^f \sim_{i_0}^f u_1^f, u_1^f \sim_{i_1}^f u_2^f, \dots, u_n^f \sim_{i_n}^f t^f$ and $\{i_0, i_1, \dots, i_n\} \subseteq G$. We will perform induction on the length n of this path.

For the base case of our inductive proof, let $n = 0$. Thus, we consider a path $\langle s^f, \sim_{i_0}^f, t^f \rangle$, matching the base case of our Lean implementation:

```
| done (hi : i ∈ G) (hst : t ∈ m.f.rel i s) : C_path G s t
```

Thus we need to prove that $M, t \models \psi$, given that $s^f \sim_{i_0}^f t^f$, for some $i_0 \in G$. This base case differs from a more traditional inductive proof on a common knowledge path, like the proof by Ågotnes and Alechina [1], where the base case is simply the starting state, with the path being $\langle s^f \rangle$. Note that this is equivalent to our base case with the additional assumption that $s^f = t^f$, as we must by reflexivity have $s^f \sim_{i_0}^f s^f$.

Next our inductive step needs to match our recursive case in Lean:

```
| next (hi : i ∈ G) (hsu : u ∈ m.f.rel i s) (ih : C_path G u t) :
  C_path G s t
```

Here we build the path recursively from the front: so when looking at the path from s^f to t^f , we consider first the individual knowledge relation from s^f to the second state in the path. Then we recursively define the rest of the path from the second state to t^f . Our inductive step must match this format. Let the first state between s^f and t^f be u^f , where $s^f \sim_i^f u^f$ for some $i \in G$, and let the common knowledge path for group G of length n be $\langle u^f, \sim_{i_0}^f, u_1^f, \sim_{i_1}^f, u_2^f, \dots, u_n^f, \sim_{i_n}^f, t^f \rangle$. The inductive hypothesis states that if $C_G\psi \in u^f$, then $M^f, t^f \models \psi$. Again, this approach to the inductive step is different from the more usual inductive proof on a common knowledge path by Ågotnes and Alechina [1]. In their case the inductive step splits a path of length $n + 1$ into a path from the starting state (s^f) to the n th state in the path, and a single knowledge relation from the n th to the end state (t^f).

Note that for our inductive proof on the common knowledge path, both in the base case and in the inductive case we need to prove something (that ψ holds in the base case, that it contains $C_G\psi$ for the inductive step) about a state (t^f for the base case, w^f for the inductive step) which is connected from s^f by an individual knowledge relation for some agent in G . Thus we now show that for any state w^f , where there is a relation $s^f \sim_j^f w^f$ for some $j \in G$, we must have both $M^f, w^f \models \psi$ and $C_G\psi \in w^f$. From $C_G\psi \in s^f$ we must have $K_j(C_G\psi) \in s^f$ by definition of $cl(C_G\psi)$, propositional logic, and axioms **(C)**, **(K)** and **(RN)**. Thus by definition of \sim_i^f we must also have $K_j(C_G\psi) \in w^f$. Then we must also have $C_G\psi \in w^f$ by axiom **(T)**. Hereby we have proven $M^f, t^f \models \psi$ for the inductive step in our proof. Additionally, from $C_G\psi \in w^f$, we know $\psi \in w^f$, by axioms **(T)**, **(C)**, **(K)** and **(RN)**. Note that by the inductive hypothesis for the truth lemma ($\forall s \in S^C, M^f, s^f \models \psi \leftrightarrow \psi \in s^f$), we must then also have $M^f, w^f \models \psi$. Therefore we have proven $M^f, t^f \models \psi$ for the base case in our proof.

8.5 Finalizing the completeness proof

It remains to prove the final theorem:

► **Theorem 6** (Completeness of CLC [1, Corollary 1]). $\forall \varphi, \models \varphi \Rightarrow \vdash \varphi$

We prove the contrapositive by showing that every formula not provable by CLC is not globally valid: $\not\vdash \varphi \Rightarrow \not\models \varphi$. From $\neg \vdash \varphi$ we know that $\{\neg\varphi\}$ must be a consistent set. By Lindenbaum's lemma [24] the set can thus be extended into some maximally consistent set Σ that is equal to some state $s \in S^C$. Note that when filtered through $cl(\varphi)$, we will still have $\neg\varphi \in s^f$. By Lemma 5 $\neg\varphi$ is true in that filtered state, and thus φ is not. Thus φ is not globally valid.

We have thus verified the proof theory and model theory of CLC relate to each other as expected by proving both soundness and completeness. All Lean lemmas and definitions about (filtered) canonical model construction can be reused to prove that CL and CLK are also sound and complete (see [21] for details). For CL, as mentioned previously, this is done by proving the truth lemma for the canonical coalition model for CL. For Coalition Logic with individual knowledge (CLK), the proofs are analogous to the proofs presented here, omitting any parts related to common knowledge.

9 Conclusion and Discussion

In this paper, we have described the successful implementation of soundness and completeness proofs for CLC in Lean. Our project consists of approximately 6,000 lines of code. Of these, approximately 300 lines are specific to Coalition Logic (CL), 700 are specific to Coalition Logic with Knowledge operators (CLK), and 1,100 to Coalition Logic with Common knowledge

operators (CLC). The remaining almost 4,000 lines are shared between the three. In addition, we make extensive use of the Lean mathematical library Mathlib. We will not mention a De Bruijn factor for our development, as there is no direct comparison possible between the scope of our work and any of the relevant papers.

Much of the complexity of our formalization comes from the need to deal with finiteness in Lean. To access properties of finiteness in Lean, we needed to use specific data types. This is most notable in our formalization of ϕ_X , where we create three different definitions for when X is a `Set` a `Finset` (finite set) and a `List`. Of the three mentioned data types only the `List` is ordered in Lean (in our case, when converting from a (finite) set the order is arbitrary) and therefore allows us to iterate over elements. However when translating some (finite) set into a list we often need to keep track of relevant properties about the initial `Set`. For instance, we may need to remember that our resulting `List` contains no repeating elements. We are therefore often required to create separate lemmas for each data type, and manually pass such information forward. These translations consequently add a lot of work. However, each individual step was relatively simple with the existing Mathlib library [25]. Additionally, some of these challenges are likely exacerbated by our goal to keep our Lean proofs reasonably similar to their respective paper proofs. For instance, in our formalization we define finite conjunctions and disjunctions recursively. However to show a finite conjunction is provable or is contained in some state, we simply need to show that all conjuncts are provable or are contained within that state. Similarly for finite disjunctions we aim to show that one disjunct is provable or contained within the state. Thus a deeper embedding using Lean’s native \forall and \exists quantifiers may have been more natural.

Another difficulty with formalization is that there are many trivial lemmas that need detailed proofs in Lean, which makes formalization cumbersome and time-consuming. This is especially notable with the lemmas about the finite closure (cl), for instance that it is closed under single negations. Despite being trivial by our definition of cl , the proof in Lean is long because of how many cases need to be considered. This highlights the need for continued work on increasing automation in Lean. Specifically, these long but trivial inductive proofs would be ideal candidates for better automation.

Despite these challenges, one of the main advantages of formalizing this proof is that it required us to be precise about exactly when we were using hypotheses and assumptions. In our case, this led to us easily showing that the completeness proof for CLC described by Ågotnes and Alechina [1] also holds if we extend the syntax to also allow formulas of the form $C_{\emptyset}\varphi$. Programmatic formalization lends itself well to these tests of generalization: it automates the work of re-checking an entire proof every time a hypothesis is slightly changed or removed [4, 10].

Aside from dealing with the nature of formalization itself, one of the goals of our research was to allow for reuse of lemmas and definitions across different logics. To this end we introduced Lean `classes` for each logics syntax and axiomatic system. Importantly, we were able to define the canonical model M^C for these classes, such that the model can be built for CL and any of its extensions. Additionally we provided a large number of lemmas defined using our classes for propositional logic, CL, CLK and CLC. We hope that an increasing library of these kinds of proofs can aid future research into formalizing modal logics, especially work on formalizing the other types of Epistemic Coalition Logic described by Ågotnes and Alechina [1].

We note, however, that we did not add semantics to our class definitions. This choice was made as the semantics are only used in inductive proofs. We could not use classes for inductive proofs, as they act as minimum requirements for the syntax and proof system of

the logic. However, each individual case in an inductive proof could be separated into its own lemma if the semantics was added to the generic classes. Future work could thus look into expanding our classes and creating such generic proofs. Even more interesting would be to define the logics in such a way that we can use generic data structures for inductive proofs.

References

- 1 Thomas Ågotnes and Natasha Alechina. Coalition logic with individual, distributed and common knowledge. *Journal of Logic and Computation*, 29(7):1041–1069, 2019. doi:10.1093/logcom/exv085.
- 2 Mussab Alaa, Aos Alaa Zaidan, Bilal Bahaa Zaidan, Mohammed Talal, and Miss Laiha Mat Kiah. A review of smart home applications based on internet of things. *Journal of Network and Computer Applications*, 97:48–65, 2017. doi:10.1016/j.jnca.2017.08.017.
- 3 Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in lean 4. Accessed 2024-01-18. URL: https://lean-lang.org/theorem_proving_in_lean4/.
- 4 Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, Johannes Hölzl, and Jannis Limperg. *The Hitchhiker’s Guide to Logical Verification*. Vrije Universiteit Amsterdam, 2021 edition, 2021.
- 5 Colm Baston and Venanzio Capretta. Game forms for coalition effectivity functions. In *TYPES 2019*, pages 26–27, 2019. URL: <https://nottingham-repository.worktribe.com/output/2681068>.
- 6 Bruno Bentzen. A Henkin-style completeness proof for the modal logic S5. In *Lecture Notes in Computer Science*, pages 459–467. Springer International Publishing, 2021. doi:10.1007/978-3-030-89391-0_25.
- 7 Lubor Budaj. Formalization of modal logic S5 in the Coq proof assistant, 2022. Bachelor’s Thesis, University of Groningen. URL: https://fse.studenttheses.ub.rug.nl/28482/1/BSc_Thesis_final.pdf.
- 8 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9195, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.
- 9 Asta Halkjær From. Formalized soundness and completeness of epistemic logic. In Alexandra Silva, Renata Wassermann, and Ruy de Queiroz, editors, *Logic, Language, Information, and Computation*, pages 1–15, Cham, 2021. Springer International Publishing.
- 10 Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34:3–25, 2009.
- 11 Balaji Parasumanna Gokulan and Dipti Srinivasan. An introduction to multi-agent systems. In Dipti Srinivasan and Lakhmi C. Jain, editors, *Innovations in Multi-Agent Systems and Applications*, pages 1–27. Springer, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-14435-6_1.
- 12 Valentin Goranko, Wojciech Jamroga, and Paolo Turrini. Strategic games and truly playable effectivity functions. *Autonomous Agents and Multi-Agent Systems*, 26(2):288–314, March 2013. doi:10.1007/s10458-012-9192-y.
- 13 Frans C. A. Groen, Matthijs T. J. Spaan, Jelle R. Kok, and Gregor Pavlin. Real world multi-agent systems: Information sharing, coordination and planning. In *TbiLLC ’05*, volume 4363 of *LNCS*, pages 154–165, Cham, 2007. Springer. doi:10.1007/978-3-540-75144-1_12.
- 14 Joni Helin. Combining deep and shallow embeddings. *Electronic Notes in Theoretical Computer Science*, 164(2):61–79, 2006. doi:10.1016/j.entcs.2006.10.005.
- 15 Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal methods in safety-critical railway systems. In *10th Brazilian symposium on formal methods*, pages 29–31, August 2007.
- 16 Pierre Lescanne and Jérôme Puissegur. Dynamic logic of common knowledge in a proof assistant. (preprint), 2007. doi:10.48550/arXiv.0712.3146.
- 17 Jiayu Li. Formalization of pal-s5 in proof assistant. (preprint), 2020. doi:10.48550/arXiv.2012.09388.

- 18 Cláudia Nalon, Lan Zhang, Clare Dixon, and Ullrich Hustadt. A resolution prover for coalition logic. In Fabio Mogavero, Aniello Murano, and Moshe Y. Vardi, editors, *Proceedings 2nd International Workshop on Strategic Reasoning, SR 2014, Grenoble, France, April 5-6, 2014*, volume 146 of *EPTCS*, pages 65–73, 2014. doi:10.4204/EPTCS.146.9.
- 19 Paula Neeley. A formalization of dynamic epistemic logic. Master’s thesis, Carnegie Mellon University, 2021.
- 20 Kai Obendrauf. CL-Lean Formalization. Software, swhId: swh:1:dir:5679444e6dc3b26cd7f1c54786bff9be89541c19 (visited on 2024-07-08). URL: <https://github.com/kaiobendrauf/cl-lean>.
- 21 Kai Obendrauf. Formalizing completeness proofs for coalition logic with and without common knowledge in Lean. Master’s thesis, Vrije Universiteit Amsterdam, 2023. doi:10.5281/zenodo.12582708.
- 22 Marc Pauly. A modal logic for coalitional power in games. *Journal of Logic and Computation*, 12(1):149–166, 2002. doi:10.1093/logcom/12.1.149.
- 23 Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *CoRR*, abs/2001.04301, 2020. arXiv:2001.04301.
- 24 Alfred Tarski. Über einige fundamentale Begriffe der Metamathematik. *Sprawozdania z Posiedzeń Towarzystwa Naukowego Warszawskiego. Wydział III*, 23:22–29, 1930.
- 25 The mathlib Community. The Lean mathematical library. In *CPP 2020*, pages 367–381, New York, NY, USA, 2020. ACM. doi:10.1145/3372885.3373824.
- 26 Johan Van Benthem. Games in dynamic-epistemic logic. *Bulletin of Economic Research*, 53(4):219–248, 2001. doi:10.1111/1467-8586.00133.
- 27 Wiebe van der Hoek and Michael J. Wooldridge. Logics for multiagent systems. *AI Mag.*, 33(3):92–105, 2012. doi:10.1609/aimag.v33i3.2427.
- 28 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL ’89, Principles of Programming Languages*, pages 60–76. ACM, 1989. doi:10.1145/75277.75283.