Annotated Systems for Common Knowledge

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Ricardo Wehbe
von Argentinien

Leiter der Arbeit:
Prof. Dr. G. Jäger
Institut für Informatik und angewandte Mathematik

Annotated Systems for Common Knowledge

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Ricardo Wehbe
von Argentinien

Leiter der Arbeit:
Prof. Dr. G. Jäger
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 24. Oktober 2010

Der Dekan:
Prof. Dr. S. Decurtins

Acknowledgements

First of all I am deeply indebted to Prof. Gerhard Jäger, who gave me the opportunity to work in this wonderful group. His deep insight will remain for me an influence which will go far beyond this work.

I want also to thank Prof. Sommaruga for having taken the time to read this thesis. I acknowledge the generous support given me by the Hasler Foundation, and to I am also very grateful to Kai Brünnler for the time (and patience!) he dedicated to reviewing this work. I am also grateful to Prof. Thomas Strahm.

Finally, the whole TIL group provided the ideal human landscape. I will miss our coffee-breaks!

A María Inés.

Contents

Chapter 1

Prologue

The logic of Common Knowledge [28] is a multimodal logic [18] in which the modalities are interpreted as knowledge of the agents about a universe of discurse and about the knowledge of other agents. It is thus related to Epistemology, as explained in Chapter 2. Our work is also related to the search of cut-free systems for modal logics, notably to the work of Kai Brünnler and Martin Lange on annotated systems for temporal logics [13], which in turn related to the work of Colin Stirling and Martin Lange on Focus Games [56]. The adaptation made to the system of Brünnler and Lange for Common Knowledge is described in Chapter 3.

A prototype for an implementation of the resulting system was written in Prolog. The proof system can be implemented in a relatively straightforward way, at least regarding the decision procedure. Some details of the implementation are discussed in Chapter 4 and the whole source code is in Appendix A. The extraction of a countermodel in the case of a non-provable sequent is a bit more complicated than the decision procedure. It is described with some detail in Appendix B. The implementation of Prolog we used (SWI- Prolog [85]) has a text interface neither allowing symbols as $\Box$, $\Diamond$, $\boxplus$ and $\diamondplus$, that are used in the language, nor capital letters as connectors (K, P, C, U.) We chose to allow the user to use these symbols and to make a translation onto normal prefix Prolog terms. Another pos-

sibility would have been to accept lowercase letters both as connectors and as atomic propositions. Appendix C explains the changes that should be done in the implementation to do this. Essentially the advantage would be that the conversion is spared. The drawback would be that the resulting form of the formulæ would still be rather difficult to read.

Chapter 5 shows various examples and compares the performance of the program under different input sequents. As in the case of tableaux [35, 78], the introduction of irrelevant branching formulæ worsens the performance. This is not surprising, since both systems are akin to each other, as shown in Chapter 6.

Chapter 2

Knowledge and Common Knowledge

Jack thinks
he does not know
what he thinks
Jill thinks
he does not know

But Jill thinks he does know it.

So Jill does not know
she does not know
that Jack does not know
that Jill thinks
that Jack does know
and Jack does not know he does not know
that Jill does not know she does not know
that Jack does not know
that Jill thinks Jack knows
what Jack thinks he does not know

Jack doesn't know he knows
and he doesn't know
Jill does not know.

Jill doesn't know she doesn't know,
and doesn't know
that Jack doesn't know he knows
and that he does not know Jill does not know.
They have no problem.

Ronald Laing, *Knots*

2.1 Introduction

In this chapter we introduce some elementary concepts about epistemic logic and common knowledge. The chapter is organised as follows: Section 2.2 contains some general concepts about epistemology. Section 2.3 presents epistemic logic and Section 2.4 discusses informally the epistemic modality we are interested in, namely common knowledge. Section 2.5 briefly discusses the original motivation. Some of the material of this chapter anticipates what is more thoroughly developed in later ones, especially in chapter 3.

2.2 Epistemology

There is little doubt that epistemological questions arise with philosophy itself. Roughly speaking, epistemology is a branch of philosophy concerned with the study of knowledge. "Knowledge" is a rather loose concept encompassing many different meanings: one knows a person, one knows the rules of chess, one knows a city. The definition of knowledge as "justified true belief", which is sometimes attributed to Plato[1] [15] seems to be no longer philosophically defensible after the famous three- page paper of Gettier [32], where a couple of counterexamples are discussed. There are cases where the possession of evidence is not sufficient for ensuring that it is not the case that a belief is true merely because of luck. A well-known example is the following one [34]:

> Jack drives through a rural area in which what appear to be barns are, with the exception of just one, mere barn façades. From the road Jack is driving on, these facades look exactly like real barns. Jack happens to be looking at the one and only real barn in the area and believes that there is a barn over there. Jack's belief is justified because his visual experience justifies his

[1] The attribution is also unclear. Gerson [31] argues that Plato rejects the basis of this analysis on the grounds that knowledge is not belief (not even "justified" or "true" belief) plus "something else."

> belief: it originates in a reliable cognitive process. Yet Jack's belief is true merely because of luck. Had he noticed one of the barn-façades instead, he would also have believed that there is a barn over there. There is agreement among epistemologists that Jack's belief does not qualify as knowledge.

The problem of the nature of knowledge appears already in the pre-Socratic fragments and it is at the background of the famous allegory of the cave in the opening of the seventh book of Plato's *Republic*. According to Gerson [31], philosophy began in Greece with the simple hypothesis that nature has an order or structure (*kosmos*) and this order is intelligible. As a consequence, it is subject to understanding (*logos*.) But it turns out that this order is not immediately evident and the nature of its undertanding constitutes already a non-trivial problem.

The idea of what is now called "positive introspection" (see Section 2.4) appears already in Thomas Aquinas and several epistemic modalities (notably those of knowledge and belief) are mentioned in the 14th century by William of Occam and incorporated into his syllogisms [11, 50].

Ancient epistemology from the beginning of Greek philosophy up to Descartes has a strong naturalistic view [31]: cognition is rooted in the understanding of the natural world. There is another, "non-naturalistic", approach: knowledge is a matter of logic and semantics and as such not a pure branch of natural science. Our work is inscribed within the latter approach.

There are some concerns about the accuracy of a representation of knowledge by means of a comparatively simple formal system [6, 7]: we have already mentioned that knowledge is a rather ample concept. We agree that some subtleties of knowledge require a more sophisticated formalism if they are formally representable at all. In this work we concentrate on knowledge of *propositions*: for instance if we say

> Jill knows that Béla Bartók wrote three piano concertos

Then we assert that Jill is an agent that is in possession of the knowledge expressed by the proposition "Béla Bartók wrote three piano concertos."

2.3 Epistemic Logic

The idea of applying the tools of formal logical analysis to epistemology is relatively modern. Up to the middle of last century the lack of an appropiate semantics limited the usefulness of epistemic logic [39]. Advances in modal logics led several researchers, notably von Wright and Hintikka, to develop a model theory for epistemic logics. The short work of von Wright *An Essay in Deontic Logic and the General Theory of Action* of 1951 [86] is recognized as having initiated the formal study of epistemic logic in its modern sense. But it was not until the publication of Hintikka's *Knowledge and Belief* [40] in the early sixties that widespread interest in epistemic logics arose, not only from philosophers but also from logicians and computer scientists[2].

The addition of a K operator to a language with the usual propositional connectives is due to Moore[3] [67], giving rise to *autoepistemic logic*. Autoepistemic logic is a nonmonotonic logic [65, 66]. Roughly speaking, a logic is nonmonotonic if a conclusion derived from certain premises may no longer be derivable if the set of premises is extended. For instance, if Γ is a set of propositional premises and φ is a propositional formula which is not a propositional consequence of Γ, then $\neg\mathsf{K}\varphi$ may be derived ("φ is not known.") But if the set of premises is extended with φ, then $\neg\mathsf{K}\varphi$ is no longer true. Nonmonotonic logics play an important rôle in commonsense reasoning. The most important ones are circumscription [63], closed world assumption [74], and default logic [75]. There is also a close relation between autoepistemic logic and Prolog, the language in which the theorem prover is implemented, through *stable model semantics* [30, 62] and especially through *negation as failure* [20], as explained in Chapter 4.

We will not work with autoepistemic logic but with explicit cognitive agents. Thus we begin with a countable set Φ of basic facts, the (*atomic propositions*). These propositions will be denoted by p, q, possibly subscripted, and a finite (and nonempty) set of agents $\mathcal{A}$, which will be denoted by 1, 2,. . . ,n.

[2]Indeed the author warns in the very first line: "The word 'logic' which occurs in the subtitle of this work is to be taken seriously."

[3]Actually he called this operator L in the original paper.

In epistemic logic it is possible to make statements of the form "1 knows p", "2 considers q possible" and "3 does not know p." Two early applications of multiagent systems to epistemic logic are [36, 57]. More complex statements are of course possible. For instance, "1 knows that he does not know p", or "2 considers possible that she knows q." A formal syntax and semantics is sketched in Section 2.4 and fully given in Chapter 3.

Possibility is not to be identified with *belief*. The distinction between knowledge and belief is a remarkably subtle one. A thorough analysis of this very interesting issue is beyond the scope of this work. We refer to the philosophical literature; see for instance the very readable book of Pollock [73] or the more technical articles of Hintikka [42] or Vorbraak [83].

The usual interpretation of knowledge and possibility in a possible-worlds setting seems a quite natural one and has been proposed by several researchers independently, among them Carnap [17], Hintikka [41], Kripke [53], and others. See [24] for a tutorial on the history of possible worlds semantics. This approach reached the form in which we know it today in the work of Kripke [54]. This interpretation has also been critisised, notably by Barwise [6, 7].

The idea is quite simple: there is a countable set of worlds, some of them having access to others (possibly to themselves.) A world w_2 is *possible* at another world w_1 if w_2 is accessible from w_1. A fact p is *known* at world w_1 if p holds at all worlds that are possible at (accessible from) w_1. In the same way, q is *possible* at world w_1 if there is at least one possible world accessible from w_1 where q holds.

The introduction of several agents makes the structure more expressive but also mode complicated: the accessibility relations between worlds are different for each agent. This reflects the fact that not all agents have the same knowledge. The epistemic statements of the form "1 knows p" or "q is possible for 2" are dependent on the world.

The various epistemic logics are based on modal logic [18]. We make a very brief survey of modal epistemic logic next. We begin with a very simple epistemic modal logic we call E. This will be a fragment of the logic we will work with in Chapter 3. We start with a *signature* $\sigma = (\Phi, \mathcal{A})$ where Φ is a countable nonempty set of *propositions* and $\mathcal{A}$ is a finite nonempty

set of *agents*. Elements of Φ are represented by p, q, possibly subscripted, and elements of $\mathcal{A}$ are represented by natural numbers starting at 1. The syntax of the logic E is given by the following grammar:

$$\varphi ::= p \,|\, \neg p \,|\, (\varphi \vee \varphi) \,|\, (\varphi \wedge \varphi) \,|\, \Box_i \varphi \,|\, \Diamond_i \varphi$$

where $p \in \Phi$ and $i \in \mathcal{A}$.

Negation of more complex formulæ are obtained by their duals and the De Morgan laws; see Chapter 3. The modality $\Box$ is intended to mean knowledge and the modality $\Diamond$ is its dual modality. Thus, the formula $\Box_i \varphi$ means "agent i knows φ" and the formula $\Diamond_i \varphi$ means "it is not the case that agent i knows $\neg\varphi$." The semantics to this logic (and to the logic we present in the next chapter) is given by *epistemic models*, defined next. We use sometimes the $\varphi \Rightarrow \psi$ (read as φ implies ψ) as an abbreviation of $(\neg\varphi \vee \psi)$.

Definition 2.1 (Epistemic frames) *Let $\sigma = (\Phi, \mathcal{A})$ be a signature. An* epistemic frame over σ *is a pair $\mathcal{F}_\sigma = (S, R)$ where S is a set of* states *or* worlds*, and $R = \{R_1, \ldots, R_n\}$ is a family of* binary relations *on S indexed by the elements of $\mathcal{A}$.*

Definition 2.2 (Epistemic Models) *Let $\sigma = (\Phi, \mathcal{A})$ be a signature. An* epistemic model over σ *is a triple $\mathcal{M}_\sigma = (S, R, v)$ where S is a set of* states*, $v : S \to \wp(\Phi)$ is a* valuation*, and $R = \{R_1, \ldots, R_n\}$ is a family of* binary relations *on S indexed by the elements of $\mathcal{A}$.*

We say that the model $\mathcal{M}_\sigma = (S, R, v)$ is *based* on the frame $\mathcal{F}_\sigma = (S, R)$. This definition and others are repeated in other chapters to make them as self-contained as possible. The semantics of the logic E is defined next. We assume henceforth that the signature $\sigma = (\Phi, \mathcal{A})$ is fixed and omit the subindices when referring to models or frames.

Definition 2.3 (Satisfaction relation) *Let $\mathcal{M} = (S, R, v)$ be a model and let $s \in S$. The* satisfaction relation $\models$ *is defined as follows:*

$$
\begin{array}{lll}
(\mathcal{M}, s) \models p & \text{iff} & p \in v(s) \\
(\mathcal{M}, s) \models \neg p & \text{iff} & p \notin v(s) \\
(\mathcal{M}, s) \models (\varphi \vee \psi) & \text{iff} & (\mathcal{M}, s) \models \varphi \text{ or } (\mathcal{M}, s) \models \psi \\
(\mathcal{M}, s) \models (\varphi \wedge \psi) & \text{iff} & (\mathcal{M}, s) \models \varphi \text{ and } (\mathcal{M}, s) \models \psi \\
(\mathcal{M}, s) \models \Box_i \varphi & \text{iff} & \text{for all } t \in S \text{ with } (s,t) \in R_i,\ (\mathcal{M}, t) \models \varphi \\
(\mathcal{M}, s) \models \Diamond_i \varphi & \text{iff} & \text{for some } t \in S \text{ with } (s,t) \in R_i,\ (\mathcal{M}, t) \models \varphi
\end{array}
$$

Given a model $\mathcal{M} = (S, R, v)$ and a state $s \in S$, a state $t \in S$ is a *possible world* for agent i at s if $(s,t) \in R_i$. We will interpret the modality $\Diamond_i$ as "agent i considers possible that." This interpretation is justified because $(\mathcal{M}, s) \models \Diamond_i \varphi$ if there is at least one possible world t at state s for agent i such that $(\mathcal{M}, t) \models \varphi$.

Example 2.4 Consider Figure 2.1. We assume that we have the signature $\sigma = (\{p, q\}, \{1\})$, i.e., there are two atomic propositions and one single agent. We indicate the atomic propositions that are true at some state in parentheses. The arrows represent the accessibility relation between the worlds for the single agent 1.

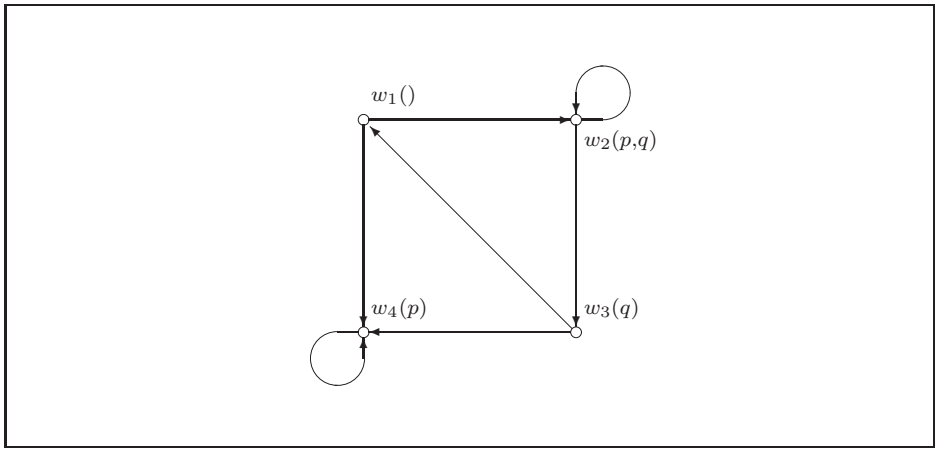


Fig. 2.1: Possible Worlds, Knowledge and Possibility.

This figure represents the model $\mathcal{M}$ where:

$$\begin{aligned} S &= \{w_1, w_2, w_3, w_4\} \\ R &= \{1 \mapsto \{(w_1, w_2), (w_1, w_4), (w_2, w_2), (w_2, w_3), (w_3, w_1), (w_3, w_4), \\ &\quad (w_4, w_4)\}\} \\ v &= \{(w_1, \varnothing), (w_2, \{p, q\}), (w_3, \{q\}), (w_4, \{p\})\} \end{aligned}$$

We have for instance:

$$\begin{aligned} (\mathcal{M}, w_1) &\models \Box_1 p \\ (\mathcal{M}, w_1) &\models \Diamond_1 q \\ (\mathcal{M}, w_2) &\models \Box_1 q \\ (\mathcal{M}, w_4) &\not\models \Box_1 q \\ (\mathcal{M}, w_1) &\models \Diamond_1 \Box_1 q \end{aligned}$$

♠

The truth or falsity of a statement clearly depends on the world at which it is formulated. The proof system S_{E} of Figure 2.2 is sound and complete for this logic.

$$\mathsf{id}\ \frac{}{\Gamma,\, p,\, \neg p} \qquad \vee\ \frac{\Gamma,\, \varphi,\, \psi}{\Gamma,\, (\varphi \vee \psi)} \qquad \wedge\ \frac{\Gamma,\, \varphi \quad \Gamma, \psi}{\Gamma,\, (\varphi \wedge \psi)}$$

$$\Box_i\ \frac{\varphi,\, \Sigma}{\Box_i \varphi,\, \Diamond_i \Sigma,\, \Delta}$$

All rules carry the proviso that the active formula in the conclusion is not part of the context.

Fig. 2.2: The system S_{E}.

The proof is in a special case of the completeness proof of Chapter 3.

The system we have presented is the so-called modal logic K. There are other modal logics which may be obtained by adding further axioms. We list in Figure 2.3 some axioms and their traditional names [18, 28].

K	$(\Box_i\varphi \wedge \Box_i(\varphi \Rightarrow \psi)) \Rightarrow \Box_i\psi$	(Distribution axiom)
T	$\Box_i\varphi \Rightarrow \varphi$	(Knowledge axiom)
4	$\Box_i\varphi \Rightarrow \Box_i\Box_i\varphi$	(Positive introspection axiom)
5	$\neg\Box_i\varphi \Rightarrow \Box_i\neg\Box_i\varphi$	(Negative introspection axiom)
D	$\neg\Box_i\bot$	(Consistency axiom)

Fig. 2.3: Some axioms of modal logic and their traditional names.

The main systems that result from adding these axioms to S_{E} are S4 (S_{E} combined with K, T, and 4) and S5 (S_{E} combined with K, T, 4, and 5.)

The axioms 4 and 5 are the introspection axioms. Axiom 4 implies that if an agent knows something, it knows that it knows it. Axiom 5 implies that if an agent does not know something, it knows that it does not know it. Axiom D implies that no agent may have inconsistent knowledge. Axiom T implies that anything that is known must be true. This strong notion of knowledge is usually what distinguishes knowledge from belief in formal systems. Axiom K is a problematic one. It is sometimes called "logical omniscience" and is implausibly strong: if an agent knows something, it knows all the logical consequences of it. According to this, any agent knowing the rules of chess knows whether there is a winning strategy for white or not.

The use of some of the axioms or not depends on the notion of knowledge one wants to capture.

There is an interesting relation between the additions of axioms and restrictions on the binary relations. The following terminology is standard (see for instance [28, 33]): A relation R on a set S is

- *reflexive* if for all $s \in S$, $(s, s) \in R$;
- *symmetric* if for all $s, t \in S$, $(s, t) \in R$ if and only if $(t, s) \in R$;

- *transitive* if for all $s, t, u \in S$, whenever $(s, t) \in R$ and $(t, u) \in R$, then $(s, u) \in R$;
- *serial* if for all $s \in S$ there is some $t \in S$ such that $(s, t) \in R$;
- *Euclidean* if for all $s, t, u \in S$, whenever $(s, t) \in R$ and $(s, u) \in R$, then $(t, u) \in R$.

We say that a family of relations has a certain property if all its relations have the property. We can establish the following correspondences between the relations R_i in a frame $\mathcal{F} = (S, R)$ and the axioms:

Axiom		Property of R
T	$\Box_i\varphi \Rightarrow \varphi$	reflexive
4	$\Box_i\varphi \Rightarrow \Box_i\Box_i\varphi$	transitive
5	$\neg\Box_i\varphi \Rightarrow \Box_i\neg\Box_i\varphi$	Euclidean
D	$\neg\Box_i\bot$	serial

We will be especially interested in one type of epistemic logic, namely *common knowledge*. This is introduced in the next section.

2.4 Armies, Children and Common Knowledge

If we say that there is common knowledge among all agents of $\mathcal{A}$ of a certain fact φ, we mean a notion that is not to be confused with the fact that all agents know φ; it goes far beyond. Technically it is said that there is common knowledge of φ among the agents of $\mathcal{A}$ when

> All agents of $\mathcal{A}$ know φ;
> All agents of $\mathcal{A}$ know that all agents know φ;
> All agents of $\mathcal{A}$ know that all agents know that all agents know φ;
> *et cætera*

In the end, this amounts to a largest fixed point. If we use the notation $\Box\varphi$ to abbreviate that all agents know φ and we denote by $\boxed{*}\varphi$ the fact that

φ is common knowledge among all agents, this can be elegantly expressed in μ-calculus [2] as follows:

$$\boxtimes^{*}\varphi = \nu X.(\Box\varphi \wedge \Box X)$$

The idea of common knowledge is not as bizarre as it may look. In fact, common knowledge arises in a number of situations. It seems to have first been mentioned in the context of convention [58]: a convention requires common knowledge of it among those who observe it. An early reference is also the multiagent system of McCarthy with a special agent called "any fool" [64]: the fact that "any fool" knows something implies that every other agent knows it too. Other areas include economics [3], protocols [84], game theory [4] and natural language, from which we give a motivating example below.

Assume the following setting, a variant of which is thoroughly discussed in [19, 72]: there is a festival with György Ligeti's music in the Colón Theatre in Buenos Aires with one major work performed each night for a week.

> **Scenario 1**: in the morning Jill reads in the early edition of the newspaper that *Atmosphères* will be played that night. Later, she sees Jack and asks, "do you know the work that will be played at the Colón tonight?"

The interesting part is the expression *the work that will be played at the Colón tonight*, which Jill intends as a reference to *Atmosphères* and under which conditions the reference will be successful. The first condition is, of course, that Jill know that *the work that will be played at the Colón tonight* uniquely describes *Atmosphères*. This is condition 1:

(1) Jill knows that the statement refers to *Atmosphères*.

The reference fails, for instance, if Jack does not have the faintest idea about the work to be played at the theatre that night. But even if he knew that, the reference might fail, as the following scenario shows.

> **Scenario 2**: in the morning Jill reads in the early edition of the newspaper that *Aventures* will be played that night and discusses the fact with Jack. Later, when he has left, she gets the late edition where a correction has been made and *Atmosphères* will be actually played that night. Later, she sees Jack and asks, "do you know the work that will be played at the Colón tonight?"

Although here condition 1 is satisfied, the reference has been made without the proper assurances. Jack has no reason to believe that the work she is referring to is *Atmosphères* and most likely he will believe it is *Aventures*. Another condition is necessary and this is the following one:

(2) Jill knows that Jack knows that the statement refers to *Atmosphères*.

Seemingly we have covered all possibilities. But still the reference might be misunderstood, as the following scenario shows:

> **Scenario 3**: in the morning Jill reads in the early edition of the newspaper that *Aventures* will be played that night and discusses the fact with Jack. When the later edition arrives, Jack sees that the work has been changed to *Atmosphères* and circles it with his red pen. Jill picks up the late edition, notes the correction and recognises Jack's mark around it. Later, she sees Jack and asks, "do you know the work that will be played at the Colón tonight?"

The last scenario satisfies conditions (1) and (2). But if Jack has no way to know that Jill has seen the late edition, he will probably still think that she is referring to *Aventures*. Another condition must be added:

(3) Jill knows that Jack knows that Jill knows that the statement refers to *Atmosphères*.

It is nevertheless still possible to think of another scenario.

> **Scenario 4**: in the morning Jill reads in the early edition of the newspaper that *Aventures* will be played that night and discusses the fact with Jack. When the later edition arrives, she notes that the work has been changed to *Atmosphères* and circles it with her green pen. Still later, as Jill watches without Jack noticing it, he picks up the late edition and sees Jill's pen mark. That afternoon, Jill sees Jack and asks, "do you know the work that will be played at the Colón tonight?"

Here conditions (1), (2), and (3) are satisfied: Jill knows that the work to be performed that night is *Atmosphères*; She knows that Jack knows it, since she saw him look at the late edition. Moreover, she knows that he knows that she knows it. Yet she is not still completely justified in thinking that the reference will be correctly interpreted. Jack might reason like this: "She knows that the work that will be performed tonight is *Atmosphères*. But since she does not know that I am already aware of the correction, she probably believes that I think it is *Aventures*." Thus, the reference is still not entirely accurate.

It is possible to fix this problem adding another condition, and there will be still possible to think of another scenario where this is insufficient. The only way to ensure that the reference is fully undestood is that both have common knowledge that it uniquely describes *Atmosphères*.

Another classical problem is the Coordinated Attack Problem [36].

> Two divisions of an army are camped on two hills overlooking a common valley. If both divisions attack the enemy simultaneously they will win the battle, whereas if only one attacks it will be defeated. The commanding general of the first division wishes to coordinate a simultaneous attack the next day. Neither general will attack unless he is sure that the other will attack with him. The generals can communicate only by means of a messenger. It takes one hour[4] for the messenger to go from

[4] Actually the time the messenger takes is irrelevant. If he would find a way to go from one camp to the other in five minutes, nothing would change.

> one camp to the other. But it is possible that he gets lost or captured by the enemy. If everything goes smoothly, how long will it take for the generals to coordinate an attack?

After having seen the first example, it is possible to guess the answer. No agreement will ever be reached. Assume General A sends a message "let's attack tomorrow at noon." He does not know whether the message was delivered or not and he will not attack until he is sure that the message has been successfully delivered. General B answers with an acknowledgment. But again, General B will not attack until he is sure that the acknowledgment has been successfully delivered. Therefore, A must send another message with an acknowledgment of the acknowledgment, and so on.

Coordinated attack deals with the problem of coordination under unreliable communications. It may be shown [28] that common knowledge is a prerequisite for coordinated attack: if the Generals attack, then it is common knowledge among them that they attack. It can also be shown that under the conditions described above, common knowledge is not attainable. It is not surprising that coordination require some degree of reliability in the communications.

Another problem that arises in distributed systems is the possibility of failure of some individual component. For instance, some processes failing during execution or some disk on a cluster breaking. This is expressed in the Byzantine Generals Problem [55, 71]. It may be stated as follows:

> Several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate only through reliable messengers (no broadcast is possible.) Some of the generals may be traitors. All generals have an initial preference (either retreat or attack) but they are all ready to retreat or attack provided the others do the same. The goal of the protocol is to reach a situation in which all loyal generals will either all attack or all retreat (the traitors may do whatever they want; there are anyway beyond control.)

This is the most general form to state the problem. Other possibilities are to restrict the actions the traitors may do: they can crash (they do not send messages any more after a certain point), they may commit omision failures sometimes, and they may act arbitrarily. A crash is a special case of the omision failure, which is in turn a special case of the so called "Byzantine failure" (arbitrary actions.) Byzantine failures are clearly the most problematic.

The problem is trivial if the set of general is guaranteed traitor-free: each general talk with each other (recall the communications are assumed to be reliable) and a common course of action is decided. The situation becomes much more complicated in the presence of traitors, since they do not know who the traitors are: a traitor might tell a loyal general to attack and another one to retreat.

As a matter of fact, this problem is quite complex. It has no solution in the case of three generals in the presence of one single traitor [55]: either the traitor initiates communication with contradictory messages, or when he receives a message, he sends an opposing one to the general who did not send the message. This situation is illustrated in figure 2.4. The traitor is represented with a black circle. For situations with more than three generals, there are algorithms for reaching an agreement when more than three quarters of the generals are loyal. The problem arises in *fault-tolerant systems*: in some redundant configurations it is important to know how many devices might fail without the intended behaviour of the system as a whole being compromised.

If the condition that whatever decision is taken it must be *simultaneously* taken by all loyal Generals, then common knowledge plays an important rôle. This may be seen in the following simple case in which we consider only crash failures (i.e., no lies): we have three Generals, no traitors, and everyone wants to retreat. They all receive a message from the others: "let us retreat." Still, they cannot take the decision because General 2 does not know whether General 3 has received the message from General 1. As in the coordinated attack problem, any decision taken by a loyal General (a non-faulty process) implies common knowledge among all loyal Generals (non-faulty processes.) In this case we have common knowledge among

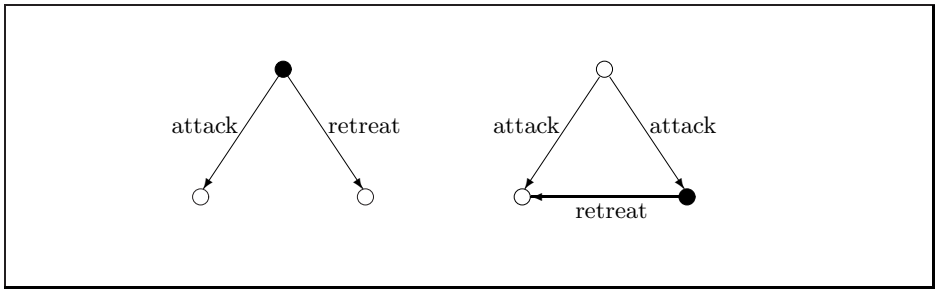


Fig. 2.4: The Byzantine Generals Problem with one traitor.

the members of a so-called *non-rigid set*, in the sense that the identity of the set is not fixed: we do not know how many Generals are loyals. The problem is thoroughly analysed in [28].

The last problem we consider is the ubiquitous muddy children puzzle [10, 28]. This puzzle appears in several different incarnations: cheating wives, cheating husbands, wise men. It may be stated in its version for three children as follows:

> A group of n children is playing. During the play k of them get mud on their foreheads. Each can see the mud on the others but not on his/her own forehead. Enters the father and says "at least one of you has mud on your forehead." The father now asks the question "does any of you know whether your forehead is clean or not?" over and over. Assuming the children are perfect reasoners, truthful and that they answer simultaneously, what happens?

Under these assumptions (strictly speaking, we need yet another assumption; we will comment on this later on), the first $k-1$ times the father asks the question the children answer "no." The k^{th} time the question is asked, the muddy children will answer "yes." The informal proof is a kind of induction on k. If $k = 1$, then the first time the father asks the question, the only muddy child, seeing that all others have immaculate foreheads, will know that he is the one. If $k = 2$, the first time the father

asks the question, both muddy children will answer "no", but the second time they may answer like this: "I see only one muddy child. Were I clean, she would have answered 'yes'; therefore I must be muddy." The argument proceeds on identical lines for $k > 1$.

There are some interesting things about this puzzle. The first one is that there is an important change in the epistemic state of the children when the father makes the announcement "at least one of you has mud on your forehead", although the information states a fact known to all the children (if $k > 1$.) The change is that what everyone knew before the announcement is common knowledge after it. The other interesting fact is that each time the question is posed and negatively answered, there is also a change in the epistemic state of the children, as explained below.

We have so far followed the presentation of Fagin et alii [28]. As we commented above, there is, though, a missing assumption. Assume the following scenario: we keep the assumptions that the children are perfect reasoners, truthful and that they answer simultaneously.

> Assume for the sake of argument that the three muddy children are the twins Stella and Brunella, who are 4 years old, and Paul, who is 5 and very proud of this fact and who, although a perfect reasoner, entertains some prejudices about the opposite sex. When the question is posed for the third time, Paul might reason as the authors of the book expect: "were I clean, the twins would each be seeing one single muddy children, namely the other twin; After the first question they should have been able to answer that they are muddy. So I must be muddy." But then he might think "but such a reasoning can be expected from a grown-up child like me and not from small children like the twins. Besides, they are girls. The fact that they don't know doesn't mean anything." And he would answer "no" again.

The last scenario shows that assuming that all children are truthful and perfect reasoners is not enough. We need a (yet) stronger assumption: the assumption must be common knowledge among them.

Now we will consider the problem from a more technical point of view. Assume there are three children, Alice, Bob and Carol. The state in which they are (clean or dirty) will be represented by a triple (a, b, c), where $a, b, c \in \{0, 1\}$. The variables a, b and c respectively represent the states of Alice, Bob and Carol. A value 0 means"clean" and a value 1 "muddy." We assume further that the actual state is $(0, 1, 1)$, i.e., Alice is clean and Bob and Carol are dirty. There are several possible worlds for each child: for instance, since Bob cannot see his own forehead, he cannot tell the world $(0, 0, 1)$ from the world $(0, 1, 1)$. We represent this situation by a graph where the nodes are labelled with all possible worlds (a, b, c). An edge labelled with $x \in \{a, b, c\}$ between two worlds means that the corresponding child cannot distinguish between them.

Before the first announcement of the father ("at least one of you has mud on your forehead") the whole situation can be depicted as in Figure 2.5.

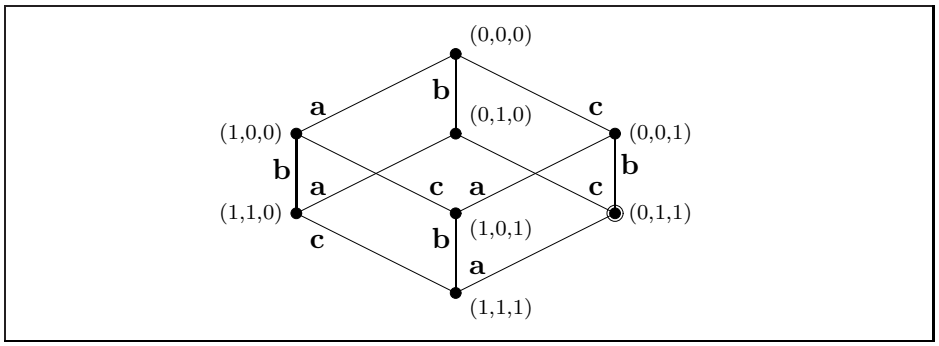


Fig. 2.5: The epistemic situation of the children before the first announcement.

The actual world is highlighted with a double circle. The representation shows what is common knowledge among the children, not the individual knowledge they have. None of the children considers for instance that the worls $(0, 0, 0)$ is possible, but this is not common knowledge yet.

What happens after the announcement of the father? It becomes common knowledge that at least one of the children is muddy. The direct consequence of it is that no child considers the world $(0, 0, 0)$ possible. This

means that there are no longer edges between that world and any other. The situation after the announcement is depicted in Figure 2.6.

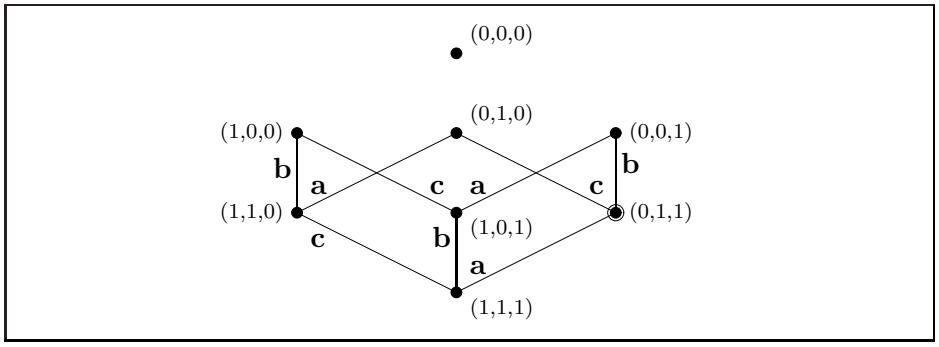


Fig. 2.6: The epistemic situation of the children after the first announcement.

Observe that in the new situation there are some worlds which are no longer uncertain for some children. For instance, the world $(1, 0, 0)$ cannot be confused with any other by Alice. This corresponds of course to the situation in which she sees only clean children and knows therefore that she is "the one." In the diagram this is reflected by the fact that there are no edges labelled with a connecting this world with any other. The same may be said of the worlds $(0, 1, 0)$ for Bob and $(0, 0, 1)$ for Carol: in the former Bob sees only clean children and in the latter it is Carol who sees only clean children. They could not confuse those world with any other.

After the father has asked the question for the first time and the children have answered "no" in unison, it is common knowledge that these three worlds are no longer possible. The new situation is depicted in Figure 2.7.

Now we have that for both Bob and Carol, the muddy children, the actual world is no longer to be confused with any other. For Alice, the clean child, it is still not possible to tell the actual world from the world $(1, 1, 1)$. When the father asks the question for the second time, Bob and Carol answer affirmatively.

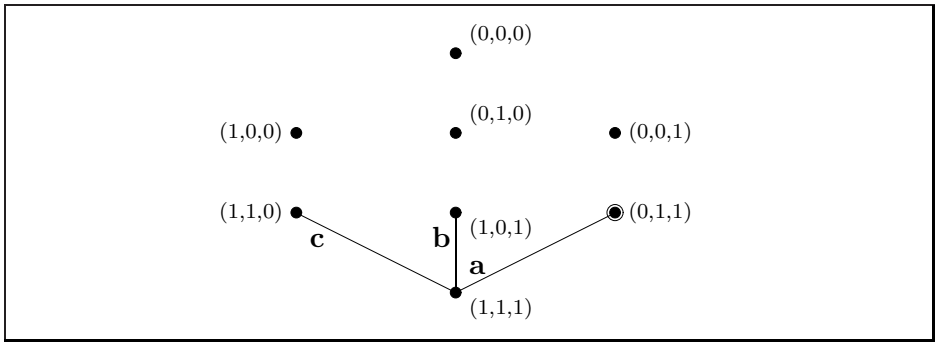


Fig. 2.7: The epistemic situation of the children after the first round of answers.

2.5 Motivations

The main motivation of the work was to obtain a cut-free proof system for common knowledge without mixing semantics and syntax as in [68], without finitising ω-rules as in [48] and without ω-rules as in [14]. The method we use is an adaptation of the annotated sequent calculus for temporal logic of Lange and Brünnler [13], which was inspired by the focus games of Lange and Stirling [56]. The idea of adapting the methodology for common knowledge was suggested by some similarities between CTL [27] and the logic of common knowledge, which we will call CK henceforth.

Chapter 3

CK with Focused Formulæ

. . . seguid vuestra historia en línea recta y nos os metáis en las curvas o transversales;
que para sacar una verdad en limpio menester son muchas pruebas y repruebas[1].

Miguel de Cervantes, *Don Quijote de la Mancha, Part II, Chapter XXVI*

3.1 Introduction

In this chapter we describe an application of the method of annotations to the logic CK [28]. The main motivation to do that is the observation of the similarities between the $\Box$ operator in CTL [27, 59] and the common knowledge operator $\boxtimes$. This suggests the application of the method of annotations, already used in CTL, to CK.

In this chapter we show, on the one hand, that the annotations method is applicable to CK and, on the other hand, that worst-case complexity is intractable, as it is for tableaux methods [1, 35, 78]. But there are some advantages: the method does not require in general the construction of the whole tree and it is a "one-pass" method, in contrast to other proof methods [27, 59].

This chapter is organised as follows. Section 3.2 contains the syntax and semantics of CK. Section 3.3 describes the rules of the proof system S_{CK}.

[1] "Go on with your story without taking any sideroads; because to extract a truth many proofs and verifications are necessary."

The soundness and completeness of the system is proved in Sections 3.4 and 3.5 respectively. Considerations on the complexity of the method are in Section 3.6 and the conclusions in Section 3.7.

3.2 Syntax and Semantics of CK

We start with a *signature* $\sigma = (\Phi, \mathcal{A})$ where Φ is a countable nonempty set of *propositions* and $\mathcal{A}$ is a finite nonempty set of *agents*. Elements of Φ will be denoted by p, q, possibly subscripted, and elements of $\mathcal{A}$ will be denoted with sequential natural numbers beginning with 1. The logic is expressed in negative normal form, i.e., only propositions appear negated. We distinguish between *formulæ* and *annotated formulæ*. The syntax of the former is given next.

Definition 3.1 (Syntax of formulæ) *The formulæ (of CK) over a signature $\sigma = (\Phi, \mathcal{A})$ are constructed according to the following grammar:*

$$\varphi ::= p \,|\, \neg p \,|\, (\varphi \vee \varphi) \,|\, (\varphi \wedge \varphi) \,|\, \Box_i \varphi \,|\, \Diamond_i \varphi \,|\, \circledast \varphi \,|\, \circledcirc \varphi$$

where $p \in \Phi$ and $i \in \mathcal{A}$.

Connectors are left-associative and the usual rules to eliminate parentheses apply. That means that $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$ denotes $((\ldots(\varphi_1 \wedge \varphi_2) \wedge \ldots) \wedge \varphi_n)$ and that $\varphi_1 \vee \varphi_2 \vee \ldots \vee \varphi_n$ denotes $((\ldots(\varphi_1 \vee \varphi_2) \vee \ldots) \vee \varphi_n)$. A *literal* is an element of the set $\Phi \cup \{\neg p \,|\, p \in \Phi\}$. Literals will be denoted by a, b, possibly subscripted. The intended meaning of formulæ with no occurrences of the symbols $\Box$, $\Diamond$, $\circledast$, or $\circledcirc$ is the usual one. The intended meaning of $\Box_i \varphi$ is that agent i *knows* φ. The intended meaning of $\circledast \varphi$ is that φ is *common knowledge* among all agents (see Chapter 2.) The connectors $\Diamond$ and $\circledcirc$ are duals of $\Box$ and $\circledast$ respectively.

We will use some abbreviations. $\top$ abbreviates $(p \vee \neg p)$ for some $p \in \Phi$ and $\bot$ abbreviates $(p \wedge \neg p)$ for some $p \in \Phi$.

Notation: if $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$ is a set of formulæ, then $\bigwedge \Gamma = \varphi_1 \wedge \ldots \wedge \varphi_n$ and $\bigvee \Gamma = \varphi_1 \vee \ldots \vee \varphi_n$. In the specific case $\Gamma = \varnothing$, then $\bigwedge \Gamma = \top$ and $\bigvee \Gamma = \bot$.

We define further the following abreviations:

$$\begin{array}{lcl}
\Box\varphi & = & \bigwedge\{\Box_i\,\varphi \mid i \in \mathcal{A}\} \\
\Diamond\varphi & = & \bigvee\{\Diamond_i\,\varphi \mid i \in \mathcal{A}\} \\
\Box_{(\leq i)}\varphi & = & \Box_1\varphi \wedge \Box_2\varphi \wedge \ldots \wedge \Box_i\,\varphi, \;\; i \geq 1 \\
\Diamond_{(\leq i)}\varphi & = & \Diamond_1\varphi \vee \Diamond_2\varphi \vee \ldots \vee \Diamond_i\,\varphi, \;\; i \geq 1
\end{array}$$

Negation of complex formulæ is inductively defined by means of their duals. Besides $\neg\neg p = p$ we have:

$$\begin{array}{lll}
\neg(\alpha \vee \beta) = (\neg\alpha \wedge \neg\beta) & \neg\Box_i\varphi = \Diamond_i\neg\varphi & \neg\boxdot\varphi = \diamondsuit\neg\varphi \\
\neg(\alpha \wedge \beta) = (\neg\alpha \vee \neg\beta) & \neg\Diamond_i\varphi = \Box_i\neg\varphi & \neg\diamondsuit\varphi = \boxdot\neg\varphi
\end{array}$$

Observe that this implies, as expected, $\neg\top = \bot$ and $\neg\bot = \top$. The semantics of formulæ is defined next.

Definition 3.2 (Epistemic Models) *Let $\sigma = (\Phi, \mathcal{A})$ be a signature. An* epistemic model over σ *is a triple $\mathcal{M}_\sigma = (S, R, v)$ where S is a set of* states*, $v : S \to \wp(\Phi)$ is a* valuation*, and $R = \{R_1, \ldots, R_n\}$ is a family of* binary relations *on S indexed by the elements of $\mathcal{A}$.*

We will talk just of *models* to refer to epistemic models. No conditions are imposed on the relations in our case. Usually the system for knowledge is S5 [28] where the relations are equivalence relations (i.e., they are reflexive, symmetric and transitive.) Sometimes the states are also called *possible worlds* [9, 28, 79]. The following notation is standard (see for instance [34].)

Notation: Assume $\mathcal{M} = (S, R, v)$ and $\mathcal{A} = \{1, \ldots, n\}$. If $s, t \in S$, then

$$\begin{array}{lll}
(s,t) \in R_{\mathcal{A}} & \text{iff} & (s,t) \in R_i \text{ for some } i \in \mathcal{A} \\
(s,s) \in R^0_{\mathcal{A}} & & \\
(s,t) \in R^1_{\mathcal{A}} & \text{iff} & (s,t) \in R_{\mathcal{A}} \\
(s,t) \in R^{q+1}_{\mathcal{A}} & \text{iff} & \exists u \in S\, ((s,u) \in R^q_{\mathcal{A}} \text{ and } (u,t) \in R_{\mathcal{A}}) \\
(s,t) \in R^+_{\mathcal{A}} & \text{iff} & (s,t) \in R^k_{\mathcal{A}} \text{ for all } k \geq 1 \\
(s,t) \in R^*_{\mathcal{A}} & \text{iff} & (s,t) \in R^k_{\mathcal{A}} \text{ for all } k \geq 0
\end{array}$$

Observe that $R_{\mathcal{A}}$ is just the union of all the relations of the family R. We will assume henceforth that the signature $\sigma = (\Phi, \mathcal{A})$ is arbitrary but fixed and omit the subindex when referring to models. In the same way, all formulæ will be assumed to be over σ.

Definition 3.3 (Paths in a model) *Let $\mathcal{M} = (S, R, v)$ be a model. A* path *in $\mathcal{M}$ is a (possibly infinite) sequence $s_0,\ s_1, \ldots$ of states of S such that for any two consecutive states s_j, s_{j+1} in the sequence, $(s_j, s_{j+1}) \in R_{\mathcal{A}}$. An* s- path *is a path with $s_0 = s$.*

Definition 3.4 (Reachability in a model $\mathcal{M}$) *Let $\mathcal{M} = (S, R, v)$ be a model and $s, t \in S$. The state t is* reachable from s in k steps *in $\mathcal{M}$ if and only if there exists a finite s-path $s_0,\ s_1, \ldots, s_k$ in $\mathcal{M}$ such that $s_k = t$. The state t is* reachable from s *if and only if it is reachable from s in k steps for some $k \geq 1$.*

Definition 3.5 *We define $\Box^j\varphi$ and $\Diamond^j$ inductively as follows:*

$$\begin{array}{lcl} \Box^0\varphi = \Diamond^0\varphi & = & \varphi; \\ \Box^{k+1}\varphi & = & \Box\Box^k\varphi; \\ \Diamond^{k+1}\varphi & = & \Diamond\Diamond^k\varphi. \end{array}$$

Any state is trivially reachable from itself in 0 step. A state s may be unreachable from itself in one step, unless $(s, s) \in R_{\mathcal{A}}$. This means that there may be a state s in a model $\mathcal{M}$ such that $(\mathcal{M}, s) \models \Box_i\varphi$ but $(\mathcal{M}, s) \not\models \varphi$. In the usual terminology of epistemic logic, agent i would not consider the actual world as a possible one. This is why most knowledge logics include the axiom T, which states that $\Box_i\varphi$ implies φ. This is equivalent to requiring the relations in R to be reflexive.

Definition 3.6 (Satisfaction relation) *Let $\mathcal{M} = (S, R, v)$ be a model and let $s \in S$. The* satisfaction relation $\models$ *is defined as follows:*

$$
\begin{array}{lll}
(\mathcal{M}, s) \models p & \text{iff} & p \in v(s) \\
(\mathcal{M}, s) \models \neg p & \text{iff} & p \notin v(s) \\
(\mathcal{M}, s) \models (\varphi \vee \psi) & \text{iff} & (\mathcal{M}, s) \models \varphi \text{ or } (\mathcal{M}, s) \models \psi \\
(\mathcal{M}, s) \models (\varphi \wedge \psi) & \text{iff} & (\mathcal{M}, s) \models \varphi \text{ and } (\mathcal{M}, s) \models \psi \\
(\mathcal{M}, s) \models \Box_i \varphi & \text{iff} & \text{for all } t \in S \text{ with } (s,t) \in R_i,\ (\mathcal{M}, t) \models \varphi \\
(\mathcal{M}, s) \models \Diamond_i \varphi & \text{iff} & \text{for some } t \in S \text{ with } (s,t) \in R_i,\ (\mathcal{M}, t) \models \varphi \\
(\mathcal{M}, s) \models \circledast \varphi & \text{iff} & \text{for all } j > 0,\ (\mathcal{M}, s) \models \Box^j \varphi \\
(\mathcal{M}, s) \models \circledcirc \varphi & \text{iff} & \text{for some } j > 0,\ (\mathcal{M}, s) \models \Diamond^j \varphi
\end{array}
$$

Proposition 3.7 *Let $\mathcal{M} = (S, R, v)$ be a model and let $s \in S$. Then*

(i) *$(\mathcal{M}, s) \models \Box^k \varphi$ if and only if $(\mathcal{M}, t) \models \varphi$ for all $t \in S$ that are reachable in k steps from s.*

(ii) *$(\mathcal{M}, s) \models \Diamond^k \varphi$ if and only if $(\mathcal{M}, t) \models \varphi$ for some $t \in S$ that is reachable in k steps from s.*

(iii) *$(\mathcal{M}, s) \models \circledast \varphi$ if and only if $(\mathcal{M}, t) \models \varphi$ for all states t that are reachable from s.*

(iv) *$(\mathcal{M}, s) \models \circledcirc \varphi$ if and only if $(\mathcal{M}, t) \models \varphi$ for some state t that is reachable from s.*

Proof. Parts (i) and (ii). Induction on k.

Base case: the base case ($k = 0$) follows directly from Definition 3.5 for (i) and (ii) in both directions.

Induction step ($\Rightarrow$). Part (i): assume $(\mathcal{M}, s) \models \Box^{k+1} \varphi$. Thus, $(\mathcal{M}, s) \models \Box\Box^k \varphi$ and therefore, for all states $t \in S$ such that $(s,t) \in R_{\mathcal{A}}$, $(\mathcal{M}, t) \models \Box^k \varphi$ and thus, by induction hypothesis, for all states $u \in S$ that are reachable from t in k steps, $(\mathcal{M}, u) \models \varphi$. But these states u are exactly the states that are reachable from s in $k+1$ steps. Part (ii): assume $(\mathcal{M}, s) \models \Diamond^{k+1} \varphi$. Thus, $(\mathcal{M}, s) \models \Diamond\Diamond^k \varphi$ and therefore, for some state $t \in S$ such that $(s,t) \in R_{\mathcal{A}}$, $(\mathcal{M}, t) \models \Diamond^k \varphi$ and thus, by induction hypothesis, for some

state $u \in S$ that is reachable from t in k steps, $(\mathcal{M}, u) \models \varphi$. But the state u is reachable from s in $k+1$ steps.

Induction step ($\Leftarrow$). Part (i): assume that $(\mathcal{M}, t) \models \varphi$ for all states $t \in S$ that are reachable from s in $k+1$ steps. Consider all states $u \in S$ such that $(s, u) \in R_{\mathcal{A}}$. Then we have that the states t are exactly those states that are reachable from the states u in k steps and therefore, by induction hypothesis, $(\mathcal{M}, u) \models \Box^k\varphi$. Hence $(\mathcal{M}, s) \models \Box\Box^k\varphi = \Box^{k+1}\varphi$. Part (ii): assume that $(\mathcal{M}, t) \models \varphi$ for some state $t \in S$ that is reachable from s in $k+1$ steps. Thus there is a state $u \in S$ such that $(s, u) \in R_{\mathcal{A}}$ and t is reachable from u in k steps. By induction hypothesis, $(\mathcal{M}, u) \models \Diamond^k\varphi$ and thus $(\mathcal{M}, s) \models \Diamond^{k+1}\varphi$.

Parts (iii) and (iv). The result follows immediately from the definition of the semantics of $\boxtimes\varphi$ and $\diamondsuit\varphi$ and parts (i) and (ii) above. ■

Besides formulæ, the language has *annotated formulæ*. An annotated formula has an *annotation* attached to it that is used to check for repetitions. A motivating example is given in section 3.3.

The syntax and semantics of annotated formulæ are defined next.

Definition 3.8 (Syntax of annotated formulæ) *Let $\sigma = (\Phi, \mathcal{A})$ be a signature, Let φ be a formula over σ and let $i \in \mathcal{A}$. The annotated formulæ of CK over σ are constructed according to the following grammar:*

$$\varphi_H ::= \boxtimes_{[H]}\varphi \mid \Box_{(\leq i)}\boxtimes_{[H]}\varphi \mid \Box_i \boxtimes_{[H]}\varphi$$

where H (the annotation*) is a finite set of finite sets of formulæ.*

Notation: we use uppercase Greek letters (Γ, Δ, Σ), possibly subscripted, to denote sets of formulæ and uppercase Latin letters (F, H), possibly subscripted, to denote sets of sets of formulæ. All the same, to avoid any possibility of confusion, we use commas to separate formulæ in a set of formulæ and vertical bars ($|$) to separate sets of formulæ in an annotation. Thus $H \,|\, \Gamma, \psi$ denotes the annotation $H \cup \{\Gamma \cup \{\psi\}\}$.

The syntax of annotated formulæ will be justified when we give the rules of the calculus.

Definition 3.9 (Corresponding formulæ) *Let* Γ *be a set of formulæ and let* $H = \{\Gamma_1, \ldots, \Gamma_q\}$ *be an annotation. Then the* corresponding formula *of* Γ *is* $\bigvee \Gamma$ *and the corresponding formula of* H *is* $\bigwedge\{\bigvee \Gamma_i \,|\, \Gamma_i \in H\}$.

Notice that the annotations $[\,]$ and $[\varnothing]$ are not the same: the former is an empty annotation whose corresponding formula is $\top$, while the latter is an annotation containing an empty set of formulæ, whose corresponding formula is $\bigwedge\{\bigvee \varnothing\} = \bigwedge\{\bot\} = \bot$.

We will not use special notations for corresponding formulæ when it is clear from the context what we are referring to. Thus, Γ may denote a set of formulæ or its corresponding formula and analogously for H.

Definition 3.10 (Presequents, sequents) *A* presequent *is a finite set of formulæ and annotated formulæ. A* sequent *is a presequent with at most one annotated formula. A sequent that consists only of formulæ (i.e., it contains no annotated formula) is called* history-free.

Notation: if $\Gamma = \{\varphi_1, \ldots, \varphi_q\}$ is a sequent and $H = \{\Gamma_1, \ldots, \Gamma_p\}$, is an annotation, then

$$\begin{aligned} \neg\Gamma = \neg \bigvee \Gamma &= \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_q \\ \neg H = \neg \bigwedge H &= \neg\Gamma_1 \vee \ldots \vee \neg\Gamma_p \end{aligned}$$

Definition 3.11 (Semantics of annotated formulæ) *Let* $\mathcal{M} = (S, R, v)$ *be a model and let* $s \in S$. *Then*

$(\mathcal{M}, s) \models \boxtimes_{[H]}\varphi$ iff for all s-paths s_0, $s_1, \ldots$ in $\mathcal{M}$

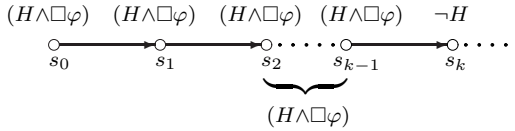
either there is a $k \geq 0$ in the path with $(\mathcal{M}, s_k) \models \neg H$ and for all j, $0 \leq j < k$, $(\mathcal{M}, s_j) \models \Box\varphi$,

or $(\mathcal{M}, s_j) \models \Box\varphi$ for all $j \geq 0$.

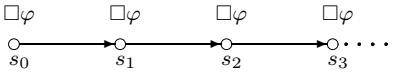
In the other cases, the semantics is extended in the natural way: if $i \in \mathcal{A}$, *we have*

$$\begin{aligned} (\mathcal{M}, s) \models \Box_i \boxtimes_{[H]}\varphi &\quad \text{iff} \quad \text{for all } t \in S \text{ with } (s,t) \in R_i, (\mathcal{M}, t) \models \boxtimes_{[H]}\varphi \\ (\mathcal{M}, s) \models \Box_{(\leq i)} \boxtimes_{[H]}\varphi &\quad \text{iff} \quad (\mathcal{M}, s) \models \Box_1 \boxtimes_{[H]}\varphi \wedge \ldots \wedge \Box_i \boxtimes_{[H]}\varphi \end{aligned}$$

Definition 3.11 is equivalent to saying that if $(\mathcal{M}, s) \models \boxtimes_{[H]}\varphi$, then all s- paths s_0, $s_1, \ldots$ have one of the following shapes:



(the rest does not matter)



(the rest does not change)

The following propositions give some characterisations of the semantics of annotated formulæ.

Proposition 3.12 *Let $\mathcal{M} = (S, R, v)$ be a model and let $s \in S$. Then $(\mathcal{M}, s) \not\models \boxtimes_{[H]}\varphi$ if and only if there is some finite s-path s_0, $s_1, \ldots, s_k$ such that:*

(i) $(\mathcal{M}, s_k) \models \Diamond\neg\varphi$

(ii) $(\mathcal{M}, s_i) \models H$ *for all states s_i in the path.*

(iii) $(\mathcal{M}, s_j) \models \Box\varphi$ *for all states with index $j < k$ in the path.*

Proof. ($\Rightarrow$) Assume $(\mathcal{M}, s) \not\models \boxtimes_{[H]}\varphi$. Then by the second disjunct of Definition 3.11, there must be an s-path $s_0, \ldots, s_k$ such that

$$(\mathcal{M}, s_k) \models \neg\Box\varphi = \Diamond\neg\varphi \tag{3.1}$$

since otherwise $\Box\varphi$ would hold for all states s_j with $j \geq 0$ and the assumption would be contradicted. If we take the least such k, then for all indices $j < k$ in the path it is the case that

$$(\mathcal{M}, s_j) \models \Box\varphi \tag{3.2}$$

Assume now that there is a state s_i in the path with $(\mathcal{M}, s_i) \models \neg H$. By (3.2), for all indices j such that $j < i$, it must be the case that

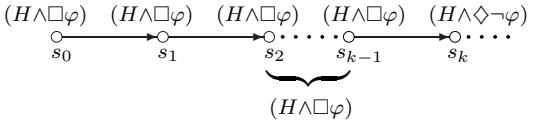
$(\mathcal{M}, s_j) \models \Box\varphi$ and by the first disjunct of Definition 3.11 we get a contradiction with the assumption. Thus for all states s_i in the path we have

$$(\mathcal{M}, s_i) \models H \tag{3.3}$$

Conditions (i), (ii), and (iii) are fulfilled by (3.1), (3.2), and (3.3), .

($\Leftarrow$) Assume there is an s-path $s_0, \ldots, s_k$ such that conditions (i), (ii), and (iii) hold. By condition (i), the second disjunct of Definition 3.11 cannot hold. By conditions (ii) and (iii), no state s_j with index $j < k$ satisfies $\neg H$, and states with indices greater than k cannot satisfy the first disjunct of Definition 3.11 because of condition (i). Thus, $(\mathcal{M}, s_0) \not\models \Box_{[H]}\varphi$. ■

Proposition 3.12 amounts to saying that, if $(\mathcal{M}, s) \not\models \circledast_{[H]}\varphi$, then there is at least one finite s-path s_0, $s_1, \ldots, s_k$ that has the following shape:



(the rest does not matter)

Definition 3.13 (Satisfiability, validity) *A formula φ is* satisfiable *if there is some model $\mathcal{M} = (S, R, v)$ such that for some state $s \in S$, $(\mathcal{M}, s) \models \varphi$. A formula φ is* valid *if for all models $\mathcal{M} = (S, R, v)$ and for all states $s \in S$, $(\mathcal{M}, s) \models \varphi$. We write in this case $\models \varphi$. A sequent is satisfiable (respectively valid) if its corresponding formula is satisfiable (respectively valid.) The definitions of satisfiability and validity for annotations and annotated formulæ are analogous.*

The semantics of annotated formulæ is similar to that of *relativised common knowledge* [9, 79]. As shown in [9], the relativised common knowledge operator cannot be expressed in the logic we are working with.

Annotated formulæ have the following fixed point expressions.

Lemma 3.14 (Fixed point expressions of annotated formulæ) *Let $\mathcal{M} = (S, R, v)$ be a model and $s \in S$. Then*

(i) $(\mathcal{M}, s) \models \circledast_{[H]}\varphi$ *if and only if* $(\mathcal{M}, s) \models \neg H \vee (\Box\varphi \wedge \Box\circledast_{[H]}\varphi)$.

(ii) $(\mathcal{M}, s) \not\models \circledast_{[H]}\varphi$ *if and only if*

(iia) $(\mathcal{M}, s) \models H$, *and*

(iib) *either* $(\mathcal{M}, s) \models \Diamond\neg\varphi$,
or there is some $t \in S$ *with* $(s, t) \in R_{\mathcal{A}}$ *such that* $(\mathcal{M}, t) \not\models \circledast_{[H]}\varphi$.

Proof. Part (i) ($\Rightarrow$). Assume

$$(\mathcal{M}, s) \models \circledast_{[H]}\varphi \tag{3.4}$$

By (3.4) and Definition 3.11, for all s-paths $s_0, s_1, \ldots$ we have one of the following cases.
(Case 1)

$$(\mathcal{M}, s_k) \models \neg H \quad \text{for some index } k \text{ in the path, and} \tag{3.5}$$
$$(\mathcal{M}, s_j) \models \Box\varphi \quad \text{for all indices } j < k \text{ in the path,} \tag{3.6}$$

(Case 2)

$$(\mathcal{M}, s_i) \models \Box\varphi \quad \text{for all indices } i \text{ in the path.} \tag{3.7}$$

If $(\mathcal{M}, s) \models \neg H$ then we are done. We consider thus the case $(\mathcal{M}, s) \models H$. By (3.6) and (3.7),

$$(\mathcal{M}, s) \models \Box\varphi \tag{3.8}$$

Take now an arbitrary s-path s_0, s_1, ... If conditions (3.5) and (3.6) hold for this path at state s_0, then they still hold at state s_1, since $k > 0$. In the same way, if (3.7) holds for this path, the condition still holds at s_1. Thus, for all states s_1 that are reachable in one step from s_0 we have

$$(\mathcal{M}, s_1) \models \circledast_{[H]}\varphi \tag{3.9}$$

By (3.9), we get that $(\mathcal{M}, s) \models \Box\circledast_{[H]}\varphi$. The other conjunct is (3.8).

Part (i) ($\Leftarrow$). Assume

$$(\mathcal{M}, s) \models \neg H \vee (\Box\varphi \wedge \Box\circledast_{[H]}\varphi) \tag{3.10}$$

We want to show that the conditions of Definition 3.11 hold. If the first part of the disjunction (3.10) holds, namely $(\mathcal{M}, s) \models \neg H$, then we are done. Assume now that the second part of the disjunction holds:

$$(\mathcal{M}, s) \models \Box\varphi \wedge \Box\circledast_{[H]}\varphi \tag{3.11}$$

By (3.11), for all s-paths $s_0, s_1, \ldots$, we have that in all cases

$$(\mathcal{M}, s_1) \models \circledast_{[H]}\varphi$$

Since $(\mathcal{M}, s) \models \Box\varphi$, we have by Definition 3.11 that $(\mathcal{M}, s) \models \circledast_{[H]}\varphi$.

Part (ii). Immediate from part (i), since this is just its negation. ■

3.3 A Proof System

The proof system S_{CK} consists of the rules shown in Figure 3.1, where the notation $\Diamond_i\Sigma$ is an abbreviation of $\{\Diamond_i\varphi \,|\, \varphi \in \Sigma\}$.

Since in all cases premises and conclusions are sequents, and thus by Definition 3.10 they contain at most one annotated formula, we have in rule $\Box_i$ that if $\Box_i\varphi$ is not focused, then Δ is history-free. Observe besides that rules $\vee$, $\diamondsuit$ and $\circledast$ can only be applied to unannotated formulæ. This is an immediate consequence of Definition 3.8, since annotated formulæ cannot appear within disjunctions, or as a subformula of $\diamondsuit$- or $\circledast$-formulæ.

The proviso that the active formula in the conclusion is not a part of the context amounts to saying that there is no hidden contraction in the system. For example, the following is not an instance of the $\vee$ rule:

$$\frac{\alpha,\ \beta,\ (\alpha \vee \beta)}{(\alpha \vee \beta)}$$

Definition 3.15 (Good instances of $\Box_i$) *Assume that the following instance of the rule $\Box_i$ is given:*

$$\Box_i \frac{\varphi,\ \Sigma}{\Box_i\varphi,\ \Diamond_i\Sigma,\ \Delta}$$

The instance of $\Box_i$ is a good instance of $\Box_i$ *if for any formula ψ, $\Diamond_i\psi \notin \Delta$.*

$$\mathsf{id}\ \frac{}{\Gamma,\, p,\, \neg p} \qquad \vee\ \frac{\Gamma,\, \varphi,\, \psi}{\Gamma,\, (\varphi \vee \psi)} \qquad \wedge\ \frac{\Gamma,\, \varphi \quad \Gamma, \psi}{\Gamma,\, (\varphi \wedge \psi)}$$

$$\boxtimes_* \frac{\Gamma,\, \Box\varphi \quad \Gamma,\, \Box\boxtimes_*\varphi}{\Gamma,\, \boxtimes_*\varphi} \qquad \diamondsuit_* \frac{\Gamma,\, \Diamond\diamondsuit_*\varphi,\, \Diamond\varphi}{\Gamma,\, \diamondsuit_*\varphi}$$

$$\Box_i\ \frac{\varphi,\, \Sigma}{\Box_i\varphi,\, \Diamond_i\Sigma,\, \Delta}\ \Gamma \text{ is history-free} \qquad \mathsf{rep}\ \frac{}{\Gamma,\, \boxtimes_{*[H|\Gamma]}\varphi}$$

$$\mathsf{foc}\ \frac{\Gamma,\, \boxtimes_{*[]}\varphi}{\Gamma,\, \boxtimes_*\varphi}\ \Gamma \text{ is history-free} \qquad \boxtimes_{*H}\ \frac{\Gamma,\, \Box\varphi \quad \Gamma,\, \Box\boxtimes_{*[H|\Gamma]}\varphi}{\Gamma,\, \boxtimes_{*[H]}\varphi}$$

All rules carry the proviso that the active formula in the conclusion is not part of the context.

Fig. 3.1: The system $S_{\mathbf{CK}}$.

If $\Box_i\varphi$ is the active formula in the conclusion of a good instance of $\Box_i$, then all formulæ of the form $\Diamond_i\psi$ in the conclusion are also active. The following definition is standard; see for example [82].

Definition 3.16 (Preproofs in a proof system S) *Let* S *be a proof system. A* preproof *in* S *is a (possibly infinite) tree whose nodes are labelled with sequents. The labels at the immediate successors of a node labelled with* Γ *are the premises by some application of a rule of* S *and* Γ *is the conclusion. A preproof of* Γ *in* S *is a preproof whose root is labelled with* Γ*.*

Definition 3.17 (Axiomatic nodes and proofs in $S_{\mathbf{CK}}$) *Let* $\mathcal{D}$ *be a preproof of* Γ *in* S_{CK}*. A node of* $\mathcal{D}$ *is* axiomatic *if it is an instance of* id *or of* rep*. If* $\mathcal{D}$ *has only finite branches and all its leaves are axiomatic, then* $\mathcal{D}$ *is a* proof *of* Γ *in* S_{CK}*. In this case we write* $S_{CK} \vdash \Gamma$*.*

A non-axiomatic node to which no rule may be applied is an *irreducible* node. The following example gives a motivation for the use of annotations. A preproof for a valid sequent is constructed without annotations, and an infinite branch appears in the preproof (which is thus no proof.) Afterwards, the use of annotations allows the construction of a proof.

Example 3.18 Assume for simplicity that $\mathcal{A} = \{1\}$ (the set of agents is a singleton.) Let us first try to construct a proof for $\Gamma = \{\circledast\neg p, \boxtimes p\}$, which is clearly valid, without using annotations. This attempt is shown in Figure 3.2.

Recall that we are using the system for proof-search and therefore the preproof progresses bottom-up. Active formulæ are underlined in the conclusions, except in the case of axioms and the $\Box_1$ rule.

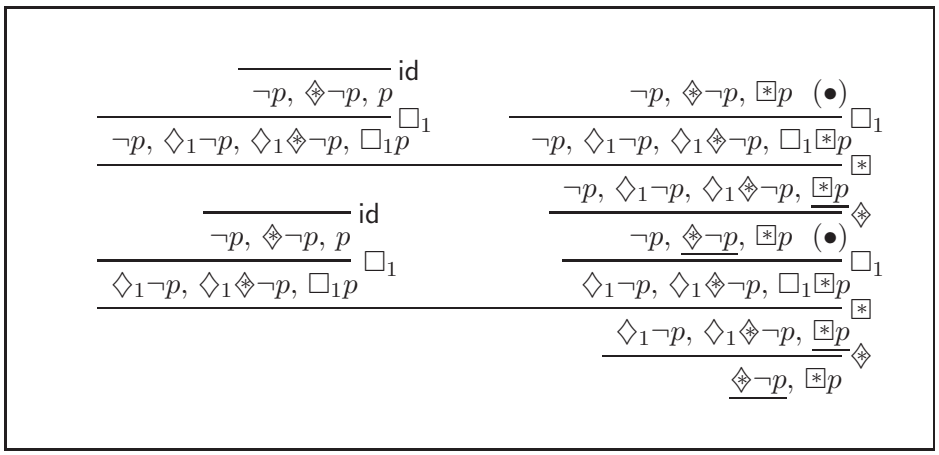


Fig. 3.2: A preproof of a valid sequent without annotations.

The sequents marked with $(\bullet)$ are repeated. If we go on, the right branch will be infinite; thus the proof cannot be constructed this way. Since the common knowledge operator $\boxtimes$ is a largest fixed point, this is an instance of a "good repeat" [56]. The proof using annotations is shown in Figure 3.3.

In this case, we have got a proof for the sequent. ♠

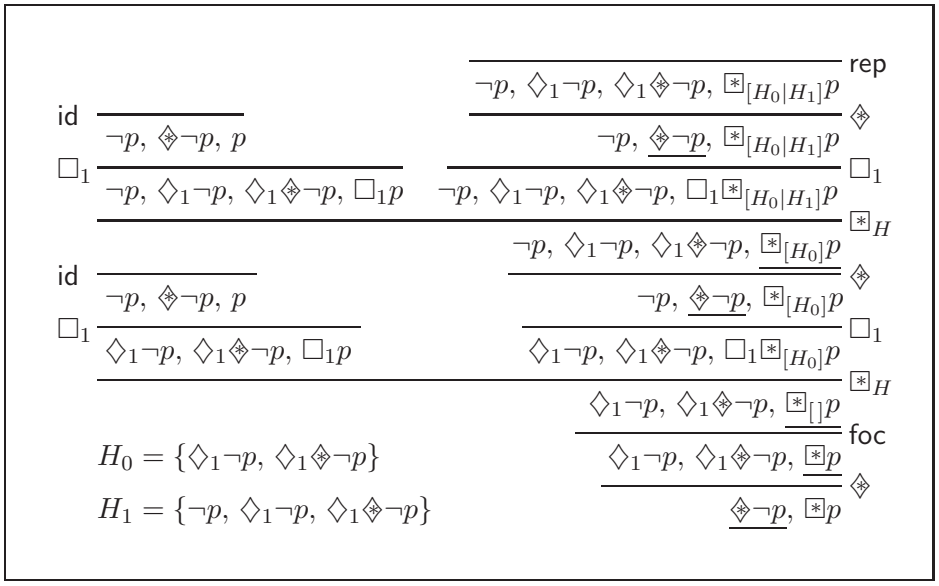


Fig. 3.3: A proof of a valid sequent using annotations.

3.4 Soundness

In this section we give two definitions of soundness, a weak and a strong one. We prove that all rules of S_{CK} but two are strongly sound, the only exception being $\Box_i$ and $\boxed{*}_H$, which are weakly sound but not strongly sound. The system S_{CK} is thus weakly sound.

Definition 3.19 (Strong and weak soundness) *A rule is said to be* strongly sound *if, for all models $\mathcal{M} = (S, R, v)$, for all states $s \in S$, if $(\mathcal{M}, s)$ satisfies the premises, then it satisfies the conclusion.*

A rule is said to be weakly sound *if the validity of its premises implies the validity of its conclusion.*

We will say that a rule is *sound* if it is weakly sound. Strong soundness clearly implies weak soundness: if for any model $\mathcal{M} = (S, R, v)$, for any $s \in S$, the fact that $(\mathcal{M}, s)$ satisfies the premises implies that it satisfies

the conclusion, if the premises are valid then the conclusion must also be valid. The converse is not true: see the proof of Lemma 3.22 for some counterexamples.

Fact 3.20 *Let $\mathcal{M} = (S, R, v)$ be a model, let $s \in S$ and let φ be a formula. Then $(\mathcal{M}, s) \models \boxtimes\varphi$ if and only if $(\mathcal{M}, s) \models \Box\varphi \wedge \Box\boxtimes\varphi$.*

Proof.

$$\begin{array}{rll}
(\mathcal{M}, s) \models \boxtimes\varphi & \text{iff} & \text{for all } j > 0,\ (\mathcal{M}, s) \models \Box^j\varphi \\
& \text{iff} & (\mathcal{M}, s) \models \Box\varphi \text{ and for all } j > 1,\ (\mathcal{M}, s) \models \Box^j\varphi \\
& \text{iff} & (\mathcal{M}, s) \models \Box\varphi \text{ and for all } j > 0,\ (\mathcal{M}, s) \models \Box\Box^j\varphi \\
& \text{iff} & (\mathcal{M}, s) \models \Box\varphi \text{ and } (\mathcal{M}, s) \models \Box\boxtimes\varphi \\
& \text{iff} & (\mathcal{M}, s) \models \Box\varphi \wedge \Box\boxtimes\varphi
\end{array}$$

■

In Lemma 3.21 below, we fix a model $\mathcal{M} = (S, R, v)$. The idea of the proof is to assume the existence of a state $s \in S$ such that $(\mathcal{M}, s)$ satisfies the premises but not the conclusion and to show that that assumption leads to a contradiction, thus showing that if $(\mathcal{M}, s)$ satisfies the premises it must also satisfy the conclusion.

Lemma 3.21 (Strong soundness) *Rules* id*,* $\vee$*,* $\wedge$*,* $\diamondsuit$*,* $\boxtimes$*,* foc*, and* rep *are strongly sound.*

Proof. Strong soundness is obvious for rules id, $\vee$, and $\wedge$. It is also obvious for rule foc (recall that the corresponding formula of an empty annotation is $\top$.) We consider the other cases.

Rule $\diamondsuit$. Assume:

$$(\mathcal{M}, s) \models \Gamma, \Diamond\diamondsuit\varphi, \Diamond\varphi, \tag{3.12}$$

$$(\mathcal{M}, s) \not\models \Gamma, \diamondsuit\varphi. \tag{3.13}$$

By (3.13), we have $(\mathcal{M}, s) \models \neg\Gamma \wedge \boxtimes\neg\varphi$ and by Fact 3.20, we get:

$$(\mathcal{M}, s) \models \neg\Gamma \wedge \Box\neg\varphi \wedge \Box\boxtimes\neg\varphi \tag{3.14}$$

Expressions (3.12) and (3.14) are contradictory.

Rule $\circledast$. Assume

$$\begin{aligned}(\mathcal{M}, s) &\models \Gamma, \Box\varphi, && (3.15)\\ (\mathcal{M}, s) &\models \Gamma, \Box\circledast\varphi, && (3.16)\\ (\mathcal{M}, s) &\not\models \Gamma, \circledast\varphi. && (3.17)\end{aligned}$$

By (3.17), $(\mathcal{M}, s) \models \neg\Gamma$. Thus by (3.15) and (3.16), $(\mathcal{M}, s) \models (\Box\varphi \wedge \Box\circledast\varphi)$. By Fact 3.20, this contradicts (3.17).

Rule rep. Assume $(\mathcal{M}, s) \not\models \Gamma, \circledast_{[H|\Gamma]}\varphi$. Thus we have, on the one hand

$$(\mathcal{M}, s) \models \neg\Gamma \tag{3.18}$$

and, on the other hand, $(\mathcal{M}, s) \not\models \circledast_{[H|\Gamma]}\varphi$. By Lemma 3.14 part (iia), and keeping in mind that the annotation $[H|\Gamma]$ is interpreted as the conjunction of the corresponding formula of H and Γ, we get:

$$(\mathcal{M}, s) \models H \wedge \Gamma \tag{3.19}$$

Expressions (3.18) and (3.19) are contradictory. ■

Lemma 3.22 (Weak soundness) *Rules $\Box_i$ and $\circledast_H$ are (weakly) sound but not strongly sound.*

Proof. We prove first weak soundness.

Rule $\Box_i$. Assume

$$\begin{aligned}&\models \varphi, \Sigma, && (3.20)\\ &\not\models \Box_i\varphi, \Diamond_i\Sigma, \Delta. && (3.21)\end{aligned}$$

If $\Sigma = \{\psi_1, \ldots, \psi_q\}$, we have by (3.21) that there must be a model $\mathcal{M} = (S, R, v)$ with a state $s \in S$ such that

$$\begin{aligned}(\mathcal{M}, s) \models\ & \neg\Box_i\varphi \wedge \neg\Diamond_i\psi_1 \wedge \ldots \wedge \neg\Diamond_i\psi_q \wedge \neg\Delta\\ &= \Diamond_i\neg\varphi \wedge \Box_i\neg\psi_1 \wedge \ldots \wedge \Box_i\neg\psi_q \wedge \neg\Delta \qquad (3.22)\end{aligned}$$

The satisfiability of $\Diamond_i \neg\varphi$ in (3.22) implies that there is some $t \in S$ such that $(s,t) \in R_i$ and $(\mathcal{M},t) \models \neg\varphi$. The conjuncts $\Box_i \neg\psi_j$ in (3.22) imply that for all states $u \in S$ such that $(s,u) \in R_i$, it is the case that $(\mathcal{M},u) \models \neg\Sigma$. In particular, $(\mathcal{M},t) \models \neg\varphi \wedge \neg\Sigma$, which contradicts (3.20).

Rule $\boxtimes_H$. This is the most interesting case. Assume

$$\models \quad \Gamma, \Box\varphi, \tag{3.23}$$

$$\models \quad \Gamma, \Box\boxtimes_{[H|\Gamma]}\varphi, \tag{3.24}$$

$$\not\models \quad \Gamma, \boxtimes_{[H]}\varphi. \tag{3.25}$$

By (3.25), there must be some model $\mathcal{M} = (S,R,v)$ with a state $s \in S$ such that

$$(\mathcal{M},s) \models \neg\Gamma, \quad \text{and} \tag{3.26}$$

$$(\mathcal{M},s) \not\models \boxtimes_{[H]}\varphi. \tag{3.27}$$

By (3.27) and Proposition 3.12, there is a finite s-path s_0, $s_1, \ldots, s_k$ such that

$$(\mathcal{M},s_k) \models H \wedge \Diamond\neg\varphi, \text{ and} \tag{3.28}$$

$$(\mathcal{M},s_j) \models H \wedge \Box\varphi \quad \text{for all } j,\ 0 \leq j < k. \tag{3.29}$$

Now we prove by induction on i that for all states s_i in the path, $0 \leq i \leq k$, the following assertions hold:

(i) $(\mathcal{M},s_i) \models \boxtimes_{[H|\Gamma]}\varphi$.

(ii) $(\mathcal{M},s_i) \not\models \boxtimes_{[H]}\varphi$.

Base case ($i = 0$). By (3.23), (3.24) and (3.26) we get $(\mathcal{M},s_0) \models \Box\varphi$ and $(\mathcal{M},s_0) \models \Box\boxtimes_{[H|\Gamma]}\varphi$. By Lemma 3.14 part (i), $(\mathcal{M},s_0) \models \boxtimes_{[H|\Gamma]}\varphi$, which is assertion (i), whereas assertion (ii) follows from (3.25) and (3.26).

Induction step ($i = q+1$). Assume that (i) and (ii) hold for index $i = q$ in the path. By (ii) and Lemma 3.14 part (iia), $(\mathcal{M},s_q) \models H$. Thus, by (i) and Lemma 3.14 part (i) we have

$$(\mathcal{M}, s_q) \models \neg\Gamma \vee (\Box\varphi \wedge \Box\circledast_{[H|\Gamma]}\varphi) \tag{3.30}$$

There are two possibilities to satisfy (3.30). If $(\mathcal{M}, s_q) \models \neg\Gamma$, then by (3.23) and (3.24), $(\mathcal{M}, s_q) \models \Box\varphi$ and $(\mathcal{M}, s_q) \models \Box\circledast_{[H|\Gamma]}\varphi$; the other possibility is that $(\mathcal{M}, s_q) \models \Box\varphi \wedge \Box\circledast_{[H|\Gamma]}\varphi$. In both cases, we get (i) for index $i = q+1$.

Besides, since $(\mathcal{M}, s_q) \models \Box\varphi$, if (ii) holds for index $i = q$, by Lemma 3.14 part (iib) it holds for index $i = q + 1$.

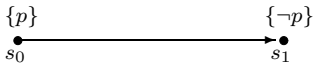
Summing up, if $(\mathcal{M}, s) \not\models \Box_{[H]}\varphi$, there must be a finite s-path $s_0, \ldots, s_k$ such that (3.28) and (3.29) hold. Besides, for all states in the path, assertions (i) and (ii) hold.

In the induction step it was shown that (i) and (ii) for a state t imply $(\mathcal{M}, t) \models \Box\varphi$. In the particular case $t = s_k$ this yields a contradiction. Therefore, there is no state s_k in the path with $(\mathcal{M}, s_k) \models \Diamond\neg\varphi$. The assumption is thus contradicted.

Now we show that the rules are not strongly sound. We assume a signature $\sigma = (\Phi, \mathcal{A})$ where $\mathcal{A} = \{1\}$ (i.e., there is one single agent.) The following is an instance of rule $\Box_1$:

$$\Box_1 \ \frac{p}{\Box_1 p}$$

Clearly $(\mathcal{M}, s) \models p$ does not imply $(\mathcal{M}, s) \models \Box_1 p$. The following figure shows a model with a state s_0 that satisfies the premise but not the conclusion.



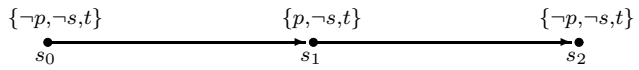
The figure corresponds to the model $\mathcal{M} = (S, R, v)$ with

- $S = \{s_0, s_1\}$.
- $R = \{1 \mapsto R_1\}$ with $R_1 = \{(s_0, s_1)\}$.
- $v(s_0) = \{p\}$, $v(s_1) = \varnothing$.

The following is an instance of rule $\boxplus_H$ for the same signature. Since the set of agents $\mathcal{A}$ is a singleton, the formulæ $\Box\varphi$ and $\Box_1\varphi$ are equivalent for this signature.

$$\boxplus_H \frac{s,\, \Box p \qquad s,\, \Box\boxplus_{[\{s\}|\{t\}]}p}{s, \boxplus_{[\{t\}]}p}$$

The following figure shows a model $\mathcal{M}$ with a state s_0 that satisfies the premises but not the conclusion.



The figure corresponds to the model $\mathcal{M} = (S, R, v)$ with

- $S = \{s_0, s_1, s_2\}$.
- $R = \{1 \mapsto R_1\}$ with $R_1 = \{(s_0, s_1), (s_1, s_2)\}$.
- $v(s_0) = \{t\}$, $v(s_1) = \{p, t\}$, $v(s_2) = \{t\}$.

On the one hand, we have that $(\mathcal{M}, s_0) \models \Box p$, since $(\mathcal{M}, s_1) \models p$. Besides, $(\mathcal{M}, s_0) \models \Box\boxplus_{[\{s\}|\{t\}]}p$ since $(\mathcal{M}, s_1) \models \boxplus_{[\{s\}|\{t\}]}p$ (because $(\mathcal{M}, s_1) \models \neg s$; see Lemma 3.14, part (i).)

Nevertheless, we have that $(\mathcal{M}, s_0) \not\models \boxplus_{[\{t\}]}p$: there is a path s_0, s_1 such that $(\mathcal{M}, s_1) \models t \wedge \Diamond\neg p$ and for all states in the path with index less than 1, namely s_0, $(\mathcal{M}, s_0) \models t \wedge \Box p$ (see Lemma 3.14, part (ii).) Observe that the premises are not valid. ■

Theorem 3.23 (Soundness of S'_{CK}) *The system S_{CK} is sound.*

Proof. Immediate from Lemmas 3.21, and 3.22. ■

3.5 Completeness

The proof of completeness is similar to the one in [13]. Completeness is proved for a more restricted system S'_{CK}, which proves extended sequents,

defined next.

Definition 3.24 (Extended sequents) *An* extended sequent *is a statement of the form* $\Gamma : l$*, where* Γ *is a sequent and* l *is a finite list of formulæ.*

A derivation in the system S_{CK} imposes no restriction on the application of the rules. In particular, any formula of the form $\boxplus\varphi$ in a history-free sequent could be focused. It would also be possible that no formula at all be focused. This may lead to non-termination, as in the first part of Example 3.3, since a bad choice could be stubbornly repeated. The preproofs in system S'_{CK} are carried out with a strategy that makes the focusing process fair. This strategy goes beyond the scope of a mere proof system, since a check for cyclic nodes is included. The list l of an extended sequent $\Gamma : l$ is used as a tool to achieve fairness in the focusing selection: it is a priority list, and only the first formula that is in the list and in Γ may be focused. After this, the priority list is changed so that the freshly focused formula gets the lowest priority (i.e., it is pushed to the end of the list.) The idea of this modification is to ensure that, when proof search is performed, every formula of the form $\boxplus\varphi$ be sooner or later focused on in a given branch.

The new system S'_{CK} is based on S_{CK}. Its rules, with the exception of $\Box'$ and foc', are the rules of S_{CK} with the addition of the priority list, which is just passed from the conclusion on to the premises in all rules but foc'.

The system S'_{CK} is intended to be exclusively used in proof-search. Thus, all applications of the rules will be backwards. Some definitions will be necessary before we give the rules of the new proof system S'_{CK}.

First we introduce the *locally reduced form* of a sequent. The idea is that a sequent is *locally reduced* if all propositional formulæ have been reduced to literals and all other formulæ are either prefixed with $\Box_i$ or with $\Diamond_i$ for some $i \in \mathcal{A}$. These latter formulæ are no longer "local", since they refer to other ("otherworldly") states. The strategy of the derivation will impose that the rule $\Box'$, which corresponds to $\Box_i$ in system S_{CK}, be only applied to a sequent when no other rule is applicable. This requires that all other rules must be applied as long as possible before rule $\Box'$ is applicable.

Definition 3.25 (Locally reduced sequents) *A sequent is* locally reduced *if it consists exclusively of literals and formulæ of the form* $\Box_i\varphi$ *and* $\Diamond_j\psi$ *for some* $i, j \in \mathcal{A}$.

In other words, no formula of the form $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $\circledast\alpha$, $\circledast_{[H]}\alpha$ or $\Diamond\!\!\!\!-\alpha$ is in a locally reduced sequent. The key observation is that there will be only one rule that will be backwards-applicable to a non-axiomatic sequent in locally reduced form, namely $\Box'$.

Example 3.26 The following sequents are locally reduced:

$$\begin{aligned}\Gamma_1 &= \{p, q, \neg q\}\\ \Gamma_2 &= \{\neg p, \Diamond_1 p, \Diamond_1\Diamond\!\!\!\!-\neg p, \Box_1\circledast_{[H]}p\}\\ \Gamma_3 &= \{\Diamond_1 p, \Diamond_2 p, \Diamond_3 p, \Diamond_1\Diamond\!\!\!\!-p, \Diamond_2\Diamond\!\!\!\!-p, \Diamond_3\Diamond\!\!\!\!-p\}\end{aligned}$$

The following sequents are not locally reduced:

$$\begin{aligned}\Delta_1 &= \{p, q, (\neg p \wedge \neg q)\}\\ \Delta_2 &= \{\Diamond\!\!\!\!-p\}\\ \Delta_3 &= \{\Diamond_1 p, \Diamond_2 p, \Diamond_3 p, (\Diamond_1\Diamond\!\!\!\!-p \vee \Diamond_2\Diamond\!\!\!\!-p), \Diamond_3\Diamond\!\!\!\!-p\}\end{aligned}$$

♠

It will be shown later on that any sequent reaches a locally reduced form after a finite number of application of the rules of S'_{CK}.

Some further restrictions will be imposed later on, to make the proof-search procedure in S'_{CK} deterministic.

The rules of system S'_{CK} are shown in Figure 3.4 .

Contrarily to the rule $\Box_i$ of system S_{CK}, the rule $\Box'$ in system S'_{CK} branches out. There is, though, an essential difference between the rule $\Box'$ and the other branching rules. In the latter, the conclusion is valid if all premises are valid. In the $\Box'$ rule, the conclusion is valid if at least one premise is valid.

Besides, there is no longer absolute freedom on the application of this rule. As explained above, the $\Box'$ rule may only be applied to sequents that

$$\mathsf{id}'\ \frac{}{\Gamma,\, p,\, \neg p\ :\ l} \qquad \vee'\ \frac{\Gamma,\, \varphi,\, \psi\ :\ l}{\Gamma,\, (\varphi \vee \psi)\ :\ l} \qquad \wedge'\ \frac{\Gamma,\, \varphi\ :\ l \quad \Gamma, \psi\ :\ l}{\Gamma,\, (\varphi \wedge \psi)\ :\ l}$$

$$\Box'\ \frac{\Sigma_1 : l \quad \dots \quad \Sigma_q : l}{\Gamma : l} \quad \text{where} \begin{cases} \Gamma \text{ is locally reduced, and} \\ \Sigma_j \in \{\Sigma_1, \dots, \Sigma_q\} \text{ iff} \\ \ \frac{\Sigma_j}{\Gamma} \text{ is a good instance of } \Box_i \end{cases}$$

$$\boxplus'\ \frac{\Gamma,\, \Box\varphi\ :\ l, \quad \Gamma,\, \Box\boxplus\varphi\ :\ l}{\Gamma,\, \boxplus\varphi\ :\ l} \qquad \diamondplus'\ \frac{\Gamma,\, \Diamond\diamondplus\varphi,\, \Diamond\varphi\ :\ l}{\Gamma,\, \diamondplus\varphi\ :\ l}$$

$$\mathsf{foc}'\ \frac{\Gamma, \boxplus_{[]}\varphi\ :\ l_1, l_2, \boxplus\varphi}{\Gamma, \boxplus\varphi\ :\ l_1, \boxplus\varphi, l_2} \quad \text{where} \begin{cases} \Gamma \text{ is history-free} \\ \text{no formula in } \Gamma \text{ occurs in } l_1 \end{cases}$$

$$\mathsf{rep}'\ \frac{}{\Gamma,\, \boxplus_{[H|\Gamma]}\varphi\ :\ l} \qquad \boxplus'_H\ \frac{\Gamma,\, \Box\varphi : l \quad \Gamma,\, \Box\boxplus_{[H|\Gamma]}\varphi : l}{\Gamma,\, \boxplus_{[H]}\varphi : l}$$

All rules carry the proviso that the active formula in the conclusion is not part of the context.

Fig. 3.4: The system $S'_{\mathbf{CK}}$.

are locally reduced. The choice of the premises is also limited, since any premise together with the conclusion must constitute a good instance of $\Box_i$.

The following definition has the goal of devicing a way to track which agents are active in each one of the multiple premises of the $\Box'$ rule.

Definition 3.27 (i-premises in a preproof) *Let $\mathcal{D}$ be a preproof in S'_{CK}. A premise $\varphi, \Sigma : l$ of $\Box'$ where the active formulæ in the conclusion Γ are $\Box_i\varphi$ and $\Diamond_i\Sigma$ is called an* i-premise *of Γ in $\mathcal{D}$.*

Observe that in general there is more than one i-premise in an instance of a $\Box'$ rule; more concretely, there is exactly one i-premise for each formula of the form $\Box_i\psi \in \Gamma$.

The next example shows how the premises of $\Box'$ are formed.

Example 3.28 Assume $\mathcal{A} = \{1, 2, 3\}$ and consider the sequent

$$\Gamma = \{\Box_1\varphi, \Box_1\psi, \Box_3\zeta, \Diamond_1\alpha, \Diamond_1\beta, \Diamond_1\gamma, \Diamond_2\theta, \Diamond_2\xi, p, q, \neg r\}$$

If $\Gamma : l$ is the conclusion of an instance of the $\Box'$ rule, the premises are

$$\begin{aligned} \Sigma_1 &= \varphi, \alpha, \beta, \gamma : l \\ \Sigma_2 &= \psi, \alpha, \beta, \gamma : l \\ \Sigma_3 &= \zeta : l \end{aligned}$$

The premises Σ_1 and Σ_2 are 1-premises of $\Gamma : l$. The premise Σ_3 is a 3- premise. There are no 2-premises. Observe that the following is an instance of rule $\Box_1$, but it is not a good instance of it. Thus, $\varphi, \alpha : l$ cannot be a premise of an instance of $\Box'$ applied to $\Gamma : l$.

$$\Box_1 \frac{\varphi, \alpha}{\Box_1\varphi, \Box_1\psi, \Box_3\zeta, \Diamond_1\alpha, \Diamond_1\beta, \Diamond_1\gamma, \Diamond_2\theta, \Diamond_2\xi, p, q, \neg r}$$

♠

Definition 3.29 (Successful and unsuccessful nodes in $S'_{\mathbf{CK}}$) *Let $\mathcal{D}$ be a finite preproof of an extended sequent $\Gamma : l$ in S'_{CK}. A node of $\mathcal{D}$ is* successful *if one of the following conditions holds:*

- *It is axiomatic (i.e., it is an instance of* id' *or of* rep'*.)*
- *It is a conclusion of a $\Box'$ rule, and at least one premise of it is successful.*
- *It is a conclusion of any other rule, and all premises of it are sucessful.*

A node in $\mathcal{D}$ is unsuccessful *if it is not successful.*

Observe that Definition 3.29 applies only to finite preproofs.

Definition 3.30 (Proofs in $S'_{\mathbf{CK}}$) *A finite preproof in S'_{CK}, is a* proof *of $\Gamma : l$ in S'_{CK} if the root is labelled $\Gamma : l$ and is successful. In this case, we write $S'_{CK} \vdash \Gamma : l$.*

There is an important difference between proofs in S_{CK} and proofs in S'_{CK} (Definitions 3.17 and 3.30): all leaves in a proof in S_{CK} must be axiomatic, while this is not necessarily the case in a proof in S'_{CK}, because of rule $\Box'$, which might have some unsuccessful premises and a successful conclusion.

Lemma 3.31 (Embedding of $S'_{\mathbf{CK}}$ in $S_{\mathbf{CK}}$) *Let Γ be a history-free sequent and let l be a list. If $S'_{CK} \vdash \Gamma : l$, then $S_{CK} \vdash \Gamma$.*

Proof. Assume $S'_{\mathrm{CK}} \vdash \Gamma : l$. Then there is a proof $\mathcal{D}'$ of $\Gamma : l$ in S'_{CK}. We construct a proof $\mathcal{D}$ of Γ in S_{CK} starting from $\mathcal{D}'$ by dropping all lists and choosing one successful premise of each instance of $\Box'$. The latter must exist because $\mathcal{D}'$ is a proof.

The proof that $\mathcal{D}$ is indeed a proof of Γ in S_{CK} is by a straightforward induction on the depth of the proof. ■

Definition 3.32 (Size of a formula, size of a sequent) *The* size *of a formula φ, denoted by $\mathsf{size}(\varphi)$, is inductively defined as follows:*

- $\mathsf{size}(a) = 1$.
- $\mathsf{size}(\alpha \wedge \beta) = \mathsf{size}(\alpha \vee \beta) = 1 + \mathsf{size}(\alpha) + \mathsf{size}(\beta)$.
- $\mathsf{size}(\Box_i \alpha) = \mathsf{size}(\Diamond_i \alpha) = \mathsf{size}(\circledast \alpha) = \mathsf{size}(\circledcirc \alpha) = 1 + \mathsf{size}(\alpha)$.

The size of a sequent $\Gamma = \{\varphi_1, \ldots, \varphi_q\}$ is

$$\mathsf{size}(\Gamma) = \mathsf{size}(\varphi_1) + \ldots + \mathsf{size}(\varphi_q)$$

Definition 3.33 (Closure of a formula) *Let φ be a formula over a signature $\sigma = (\Phi, \mathcal{A})$. The* closure of φ, *denoted by* $\mathsf{cl}_{\mathcal{A}}(\varphi)$ *is defined as follows:*

- $\mathsf{cl}_{\mathcal{A}}(a) \quad = \quad \{a\}$.
- *If* $\varphi = (\alpha \wedge \beta)$ *or* $\varphi = (\alpha \vee \beta)$, *then*
  $\mathsf{cl}_{\mathcal{A}}(\varphi) \quad = \quad \{\varphi\} \cup \mathsf{cl}_{\mathcal{A}}(\alpha) \cup \mathsf{cl}_{\mathcal{A}}(\beta)$.
- *If* $\varphi = \Box_i\alpha$ *or* $\varphi = \Diamond_i\alpha$, *then*
  $\mathsf{cl}\,(\varphi) \quad = \quad \{\varphi\} \cup \mathsf{cl}\,(\alpha)$.
- $\mathsf{cl}_{\mathcal{A}}(\boxplus\alpha) \quad = \quad \{\boxplus\alpha\} \cup \{\Box_{\leq i}\boxplus\alpha \mid i \in \mathcal{A}\} \cup \{\Box_i \boxplus\alpha \mid i \in \mathcal{A}\}$
  $\cup \{\Box_{\leq i}\alpha \mid i \in \mathcal{A}\} \cup \{\Box_i \alpha \mid i \in \mathcal{A}\} \cup \mathsf{cl}_{\mathcal{A}}(\alpha)$.
- $\mathsf{cl}_{\mathcal{A}}(\diamondplus\alpha) \quad = \quad \{\diamondplus\alpha\} \cup \{\Diamond_{\leq i}\diamondplus\alpha \mid i \in \mathcal{A}\} \cup \{\Diamond_i \diamondplus\alpha \mid i \in \mathcal{A}\}$
  $\cup \{\Diamond_{\leq i}\alpha \mid i \in \mathcal{A}\} \cup \{\Diamond_i \alpha \mid i \in \mathcal{A}\} \cup \mathsf{cl}_{\mathcal{A}}(\alpha)$.

Observe that the closure is parameterised by the set of agents. This is because the aim of Definition 3.33 is, given a preproof $\mathcal{D}$ of a formula φ, the characterisation of all the formulæ that may inhabit the nodes of $\mathcal{D}$. In the case of propositional formulæ and formulæ of the form $\Box_i\psi$ or $\Diamond_i\psi$ this is straightforward. The case of formulæ of the form $\boxplus\psi$ or $\diamondplus\psi$ is a little bit more complex. The idea is the following: A formula $\boxplus\varphi$ will have as premises $\Box\varphi$ and $\Box\boxplus\varphi$. Since $\Box\varphi$ is an abbreviation of the conjunction $\Box_1\varphi \wedge \Box_2\varphi \wedge \ldots \wedge \Box_n\varphi$, for the n agents of $\mathcal{A}$, and conjunction is left-associative, further application of rule $\wedge'$ will strip the conjunction of its rightmost conjunct. Thus we need the big conjunction $\Box\varphi = \Box_{\leq n}\varphi$ as well as all prefixes of it: $\Box_{\leq(n-1)}\varphi, \Box_{\leq(n-2)}\varphi, \ldots$. We need also the formulae of the form $\Box_i\varphi$ that are separated from the big conjunction in each application of rule $\wedge'$. The case of the other premise, $\Box\boxplus\varphi$ is analogous. The case of the formulæ of the form $\diamondplus\varphi$ is also analogous with disjunctions instead of conjunctions.

Next we provide an upper bound for the size of the closure of a formula.

Definition 3.34 (The $\delta_{\mathcal{A}}$ function) *Let φ be a formula over a signature $\sigma = (\Phi, \mathcal{A})$. The function $\delta_{\mathcal{A}}(\varphi)$ is inductively defined as follows:*

- $\delta_{\mathcal{A}}(a) = 1$.
- $\delta_{\mathcal{A}}(\alpha \wedge \beta) = \delta_{\mathcal{A}}(\alpha \vee \beta) = 1 + \delta_{\mathcal{A}}(\alpha) + \delta_{\mathcal{A}}(\beta)$.
- $\delta_{\mathcal{A}}(\Box_i \alpha) = \delta_{\mathcal{A}}(\Diamond_i \alpha) = 1 + \delta_{\mathcal{A}}(\alpha)$.
- $\delta_{\mathcal{A}}(\boxtimes \alpha) = \delta_{\mathcal{A}}(\diamondsuit \alpha) = 4 * k + \delta_{\mathcal{A}}(\alpha)$ *where* $k = |\mathcal{A}|$.

Proposition 3.35 *Let* $\mathcal{A} = \{1, \ldots, k+1\}$ *and let* $\mathcal{A}_k = \mathcal{A} \backslash \{k+1\} = \{1, \ldots, k\}$. *Then for any formula* φ, $\delta_{\mathcal{A}_k} \varphi \leq \delta_{\mathcal{A}}(\varphi)$.

Proof. Induction on φ. The base case ($\varphi = a$) and the induction steps for all formulæ other than $\varphi = \diamondsuit \psi$ or $\varphi = \boxtimes \psi$ are immediate, since the number of agents is not considered in these rules.

In the case $\varphi = \boxtimes \psi$ we have

$$\begin{array}{rcl} \delta_{\mathcal{A}_k}(\boxtimes \psi) & = & 4 * k + \delta_{\mathcal{A}_k}(\psi) \\ & \overset{\text{IH}}{\leq} & 4 * k + \delta_{\mathcal{A}}(\psi) \\ & < & 4 * k + 4 + \delta_{\mathcal{A}}(\psi) \\ & = & 4 * (k+1) + \delta_{\mathcal{A}}(\psi) \\ & = & \delta_{\mathcal{A}}(\boxtimes \psi) \end{array}$$

The case $\varphi = \diamondsuit \psi$ is analogous and omitted. ■

Observe that $\mathsf{size}(\varphi)$ and $\delta_{\mathcal{A}}(\varphi)$ differ only when φ contains subformulæ of the form $\boxtimes \psi$ or $\diamondsuit \psi$.

Lemma 3.36 (Size of the closure of a formula) *Let* φ *be a formula over a signature* $\sigma = (\Phi, \mathcal{A})$, *then the size of* $\mathsf{cl}_{\mathcal{A}}(\varphi)$ *is bounded by* $\delta_{\mathcal{A}}(\varphi)$.

Proof. Induction on $\mathsf{size}(\varphi)$ (outer induction.)

Base case (outer induction). If $\mathsf{size}(\varphi) = 1$, then $\varphi = a$ and $|\mathsf{cl}_{\mathcal{A}}(\varphi)| = |\{a\}| = 1 = \delta_{\mathcal{A}}(a)$.

Induction step (outer induction). If $\delta_{\mathcal{A}}(\varphi) = k + 1$, there are several cases to be considered.

If $\varphi = (\alpha \wedge \beta)$ or $\varphi = (\alpha \vee \beta)$, then:

$$|\mathsf{cl}_{\mathcal{A}}(\varphi)| = 1 + |\mathsf{cl}_{\mathcal{A}}(\alpha)| + |\mathsf{cl}_{\mathcal{A}}(\beta)| \overset{\mathsf{IH}}{\leq} 1 + \delta_{\mathcal{A}}(\alpha) + \delta_{\mathcal{A}}(\beta) = \delta_{\mathcal{A}}(\varphi)$$

If $\varphi = \Box_i\alpha$ or $\varphi = \Diamond_i\alpha$, then:

$$|\mathsf{cl}_{\mathcal{A}}(\varphi)| = 1 + |\mathsf{cl}_{\mathcal{A}}(\alpha)| \overset{\mathsf{IH}}{\leq} 1 + \delta_{\mathcal{A}}(\alpha) = \delta_{\mathcal{A}}(\varphi)$$

If $\varphi = \boxplus\alpha$, we apply again induction on $|\mathcal{A}|$ to prove that $|\mathsf{cl}_{\mathcal{A}}(\varphi)| \leq \delta_{\mathcal{A}}(\varphi)$ (inner induction.)

Base case (inner induction): $\mathcal{A} = \{1\}$. Then we have:

$$\begin{array}{rcl} |\mathsf{cl}_{\mathcal{A}}(\boxplus\alpha)| & = & |\{\boxplus\alpha, \Box_1\alpha, \Box_1\boxplus\alpha\}| + |\mathsf{cl}_{\mathcal{A}}(\alpha)| \\ & < & 4 * 1 + |\mathsf{cl}_{\mathcal{A}}(\alpha)| \\ & \overset{\mathsf{IH}}{\leq} & 4 * 1 + \delta_{\mathcal{A}}(\alpha) \quad \text{(outer induction hypothesis)} \\ & = & \delta_{\mathcal{A}}(\boxplus\alpha) \end{array}$$

Induction step (inner induction): $\mathcal{A} = \{1, \ldots, k, k+1\}$. We have:

$$\begin{array}{rcl} \mathsf{cl}_{\mathcal{A}}(\boxplus\alpha) & = & \{\boxplus\alpha\} \cup \{\Box_{\leq i}\boxplus\alpha \,|\, i \in \mathcal{A}\} \cup \{\Box_i \boxplus\alpha \,|\, i \in \mathcal{A}\} \\ & & \cup \{\Box_{\leq i}\alpha \,|\, i \in \mathcal{A}\} \cup \{\Box_i \alpha \,|\, i \in \mathcal{A}\} \cup \mathsf{cl}_{\mathcal{A}}(\alpha) \end{array} \tag{3.31}$$

If we use the notation $\mathcal{A}_k$ for the set $\mathcal{A}\backslash\{k+1\} = \{1, \ldots, k\}$, we have for any formula ψ:

$$\{\Box_{(\leq i)}\psi \,|\, i \in \mathcal{A}\} = \{\Box_{(\leq i)}\psi \,|\, i \in \mathcal{A}_k\} \cup \{\Box_{(\leq k+1)}\psi\} \tag{3.32}$$

This is a direct consequence of the definition of $\Box_{(\leq i)}$ in page 25. Then by (3.32) it is possible to rewrite (3.31) as follows:

$$\begin{array}{rcl} \mathsf{cl}_{\mathcal{A}}(\boxplus\alpha) & = & \boxed{\{\boxplus\ \alpha\}} \cup \boxed{\{\Box_{\leq i}\boxplus\alpha \,|\, i \in \mathcal{A}_k\}} \cup \{\Box_{\leq (k+1)}\boxplus\alpha\} \\ & & \cup \boxed{\{\Box_i \boxplus\alpha \,|\, i \in \mathcal{A}_k\}} \cup \{\Box_{k+1}\boxplus\alpha\} \\ & & \cup \boxed{\{\Box_{\leq i}\alpha \,|\, i \in \mathcal{A}_k\}} \cup \{\Box_{\leq (k+1)}\alpha\} \\ & & \cup \boxed{\{\Box_i \alpha \,|\, i \in \mathcal{A}_k\}} \cup \{\Box_{k+1}\alpha\} \cup \boxed{\mathsf{cl}_{\mathcal{A}}(\alpha)} \end{array} \tag{3.33}$$

Observe that in the last expression, the union of the framed components is $\mathsf{cl}_{\mathcal{A}_k}(\boxplus\alpha)$. Thus we can rewrite (3.33) as follows:

$$\mathsf{cl}_{\mathcal{A}}(\boxplus\alpha) = \mathsf{cl}_{\mathcal{A}_k}(\boxplus\alpha) \cup \{\Box_{\leq(k+1)}\boxplus\alpha, \Box_{k+1}\boxplus\alpha, \Box_{\leq(k+1)}\alpha, \Box_{k+1}\alpha\}$$

Hence:

$$\begin{array}{rcl} |\mathsf{cl}_{\mathcal{A}}(\boxplus\alpha)| & = & |\mathsf{cl}_{\mathcal{A}_k}(\boxplus\alpha)| + |\{\Box_{\leq(k+1)}\boxplus\alpha, \Box_{\leq(k+1)}\alpha, \Box_{k+1}\boxplus\alpha, \Box_{k+1}\alpha\}| \\ & = & |\mathsf{cl}_{\mathcal{A}_k}(\boxplus\alpha)| + 4 \\ & \overset{\mathsf{IH}}{\leq} & \delta_{\mathcal{A}_k}(\boxplus\alpha) + 4 \quad \text{(inner induction hypothesis)} \\ & = & 4 * k + \delta_{\mathcal{A}_k}(\alpha) + 4 \\ & \leq & 4 * k + \delta_{\mathcal{A}}(\alpha) + 4 \quad \text{(Proposition 3.35)} \\ & = & \delta_{\mathcal{A}}(\boxplus\alpha) \end{array}$$

The case $\varphi = \diamondplus\alpha$ is entirely analogous and is omitted. ■

Definition 3.37 (Closure of a sequent) *Let $\Gamma = \{\varphi_1, \ldots, \varphi_m\}$ be a history-free sequent over a signature $\sigma = (\Phi, \mathcal{A})$. The* closure *of Γ, denoted by $\mathsf{cl}_{\mathcal{A}}(\Gamma)$ is defined as*

$$\mathsf{cl}_{\mathcal{A}}(\Gamma) = \mathsf{cl}_{\mathcal{A}}(\varphi_1) \cup \ldots \cup \mathsf{cl}_{\mathcal{A}}(\varphi_m)$$

As a consequence of Lemma 3.36, the size of the closure of a sequent $\Gamma = \{\varphi_1, \ldots, \varphi_m\}$ is bounded by $\delta_{\mathcal{A}}(\Gamma) = \delta_{\mathcal{A}}(\varphi_1) + \ldots + \delta_{\mathcal{A}}(\varphi_m)$.

Fact 3.38

(i) *Let $\Gamma : l$ be an extended sequent with Γ history-free. Then application backwards of any rule of $\{\vee', \wedge', \Box', \diamondplus', \boxplus'\}$ yields extended sequents $\Delta : l'$, where $\Delta \subseteq \mathsf{cl}_{\mathcal{A}}(\Gamma)$.*

(ii) *Let $\mathcal{D}$ be a preproof of $\Gamma : l$ in S'_{CK}. Then for any annotated formula $\boxplus_{[H]}\varphi$ occuring in a node of $\mathcal{D}$, $H \subseteq \wp(\mathsf{cl}_{\mathcal{A}}(\Gamma))$.*

Proof. Part (i) is immediate from the rules of S'_{CK} and Definition 3.33. Part (ii) is immediate from part (i). ■

Lemma 3.39 (Finiteness of the set of labels of a preproof) *Let* $\Gamma : l$ *be an extended sequent and let* $\mathcal{D}$ *be a preproof of* $\Gamma : l$ *in* S'_{CK}*. Then the set of labels of* $\mathcal{D}$ *is finite.*

Proof. All nodes of the preproof are labelled with extended sequents $\Delta : l'$. By Fact 3.38, $\Delta \subseteq \mathsf{cl}_{\mathcal{A}}(\Gamma)$ and l' contains the same elements as l, possibly in a different order. Since the permutations of a finite list are finite, it follows by Lemma 3.36 that the number of possible labels in a preproof in S'_{CK} is finite. ■

Observe that Lemma 3.39 does not imply that the preproofs are finite. They may be infinite, but in this case they will necessarily contain repeated nodes. This property will be used to construct finite preproofs. First we need the notion of cyclicity.

Definition 3.40 (Similar sequents) *Two sequents* Γ_1 *and* Γ_2 *are similar, denoted by* $\Gamma_1 \sim \Gamma_2$*, if and only if:*

1. *they are both history-free and equal, or*
2. *they both contain annotated formulæ and* Γ_1 *is of the form* $\Gamma, \boxtimes_{[H_1]}\varphi$ *and* Γ_2 *is of the form* $\Gamma, \boxtimes_{[H_2]}\varphi$ *for some* Γ, φ*.*

Observe that, according to Definition 3.40, a history-free sequent cannot be similar to a sequent containing an annotated formula.

Definition 3.41 (Cyclic occurrences) *Let* $\mathcal{D}$ *be a preproof in* S'_{CK}*. An occurrence of an extended sequent* $\Gamma : l$ *is* cyclic *in* $\mathcal{D}$ *if there is an occurrence of* $\Sigma : l$ *as a conclusion of a* $\Box'$ *rule in the path linking the occurrence of* $\Gamma : l$ *to the root of* $\mathcal{D}$ *and* $\Gamma \sim \Sigma$*.*

The notion of cyclicity is shown in Figure 3.5.

Observe that the notion of cyclicity does not take into account annotated formulæ.

Informally, the strategy we follow in the preproofs is the following: starting from a sequent Γ, in a first step all rules except $\Box'$, foc', $\boxtimes'$ and $\boxtimes'_H$ are applied as long as possible. The resulting sequents will have the form

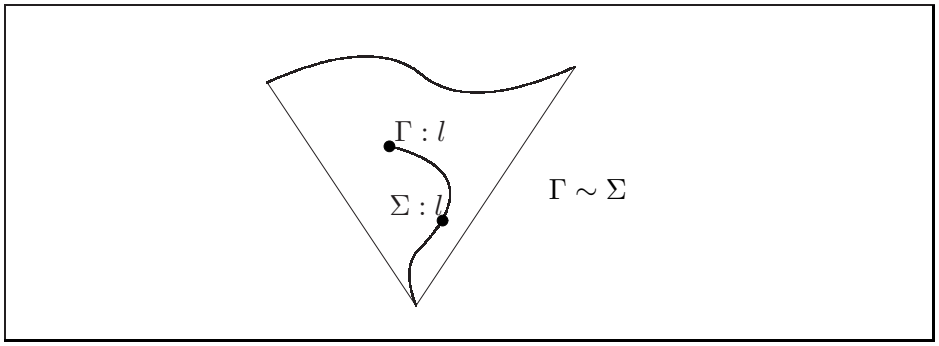


Fig. 3.5: A cyclic occurrence of $\Gamma : l$ in a preproof $\mathcal{D}$.

Σ, Δ where Σ is locally reduced and Δ consists of formulæ of the form $\boxtimes\varphi$, possibly with an occurrence of an annotated formula $\boxtimes_{[H]}\psi$. In the next step, if none of the formulæ in Δ is annotated, one of them is focused according to the priority list. Further, all rules except $\boxtimes'_H$ are applied as long as possible. The resulting sequents will have the form $\Psi, \boxtimes_{[H]}\varphi$ where Ψ is locally reduced. Finally, $\boxtimes_H$ is applied and rules $\wedge'$ are applied as long as possible.

The idea of this strategy of deferring application of rule $\boxtimes'$ is to ensure that all sequents that enter the annotations are locally reduced. This will provide a kind of "normalisation" of the sequents collected in the annotations which will be useful when we prove completeness. The application of rules $\wedge'$ at the very end (after an eventual application of rule $\boxtimes'$) is necessary to decompose the premises of rule $\boxtimes_H$.

The key point to ensure termination of this process is that whenever a cyclic node is found, the branch is closed. Thus, all cyclic occurrences will be in the leaves of the resulting tree.

The preproof is constructed be the algorithm defined next. Recall that we construct the preproof starting from the root upwards. Therefore, any application of a rule is backwards.

Algorithm 3.42 (Construction of a finite preproof in $S'_{\mathbf{CK}}$) *Let Γ be a history- free sequent and let $l(\Gamma)$ be a list of all formulæ of the form*

$\boxed{*}\varphi$ *that are in* $\mathsf{cl}_{\mathcal{A}}(\Gamma)$. *We define the algorithm of Figure 3.6 to construct a preproof* $\mathcal{D}$ *of* $\Gamma : l(\Gamma)$ *in* S'_{CK}. *In the algorithm, a non-terminal leaf is a non-cyclic leaf to which some rule may be backwards-applied.*

```
1.  input:  a history-free sequent Γ;
2.  output: a preproof 𝒟 of Γ : l(Γ) in S'_CK;
3.  begin
4.          set 𝒟 := Γ : l(Γ);
5.          while
6.                there are non-terminal leaves in 𝒟
7.          do
8.                apply id', rep', ∨', ∧', ⟐'
9                     to non-terminal leaves
10.                   until no longer possible;
11.               apply foc'to non-terminal leaves
12                    until no longer possible;
13.               apply id', rep', ∨', ∧', ⟐', □
14                    to non-terminal leaves
15.                   until no longer possible;
16.               apply □'_H to non-terminal leaves
17.                   until no longer possible;
18.               apply ∧' to non-terminal leaves
19.                   until no longer possible;
20.               apply □' to non-terminal leaves
21.                   until no longer possible;
22.         od;
23. end.
```

Fig. 3.6: Algorithm to construct a preproof in CK.

Definition 3.43 (Successful and failed runs of Algorithm 3.42) *Let* Γ *be a history-free sequent and* $l(\Gamma)$ *be a list with all formulæ of the form* $\boxed{*}\psi$ *occurring in* $\mathsf{cl}(\Gamma)$. *A run of Algorithm 3.42 with input* Γ *is* successful

if its output is a proof of $\Gamma : l\,(\Gamma)$ *in* S'_{CK}*. It is* failed *otherwise.*

The following functions are necessary to prove termination of Algorithm 3.42. As usual, we need some function whose values decrease and whose domain is a well-founded set to prove termination.

Definition 3.44 (The function δ) *Let* φ *be a formula or an annotated formula, let* $\Gamma : l$ *be an extended sequent and let* $k = |\mathcal{A}|$*. The function* $\delta(\varphi)$*, which maps formulæ into natural numbers, is defined as follows:*

- $\delta(a) = \delta(\Box_i\varphi) = \delta(\Diamond_i\varphi) = 0$ *for all* $i \in \mathcal{A}$*;*
- $\delta(\circledast\varphi) = \delta(\diamondsuit\varphi) = \delta(\circledast_{[H]}\varphi) = 2 * k$*;*
- $\delta(\varphi \vee \psi) = \delta(\varphi \wedge \psi) = 1 + \delta(\varphi) + \delta(\psi)$*.*

This function can be generalised to extended sequents in the natural way:

$$\delta(\Gamma : l) = \delta(\varphi_1) + \ldots + \delta(\varphi_m) \text{ where } \Gamma = \{\varphi_1, \ldots, \varphi_m\}$$

Fact 3.45 *Let* φ *be a formula or an annotated formula and let* $k = |\mathcal{A}|$*. Then* $\delta(\Box\varphi) = \delta(\Diamond\varphi) = k - 1$*.*

Proof. Induction on k. In the base case ($k = 1$) we have $\delta(\Box_1\varphi) = \delta(\Diamond_1\varphi) = 0 = k - 1$. For the induction step assume $k = q + 1$. Then we have:

$$\begin{array}{rcl} \delta(\Box\varphi) & = & \delta((\ldots(\Box_1 \wedge \ldots) \wedge \Box_q\varphi) \wedge \Box_{q+1}\varphi) \\ & = & \delta(\ldots(\Box_1 \wedge \ldots) \wedge \Box_q\varphi) + \delta(\Box_{q+1}\varphi) + 1 \\ & \overset{\text{IH}}{=} & q - 1 + 1 = k - 1 \end{array}$$

The case $\Diamond\varphi$ is analogous and omitted. ■

Fact 3.46

(i) *Let* $\Gamma : l$ *be the conclusion of an instance of* $\vee'$ *or* $\diamondsuit'$*, and let* $\Gamma_1 : l$ *be the premise. Then* $\delta(\Gamma) > \delta(\Gamma_1)$*.*

(ii) *Let* $\Gamma : l$ *be the conclusion of an instance of* $\wedge'$ *or* $\boxtimes'$ *and let* $\Gamma_1 : l$ *and* $\Gamma_2 : l$ *be the left and right premises respectively. Then* $\delta(\Gamma) > \delta(\Gamma_i)$ *for* $i \in \{1, 2\}$.

Proof. The cases $\vee'$ and $\wedge'$ are immediate. The case $\Gamma, \diamondsuit\varphi : l$ yields $\Gamma, \Diamond\varphi, \Diamond\diamondsuit\varphi : l$ and

$$\begin{aligned} \delta(\Gamma, \diamondsuit\varphi) &= \delta(\Gamma) + \delta(\diamondsuit\varphi) = \delta(\Gamma) + 2 * k \\ &< \delta(\Gamma) + 2 * k - 2 = \delta(\Gamma) + \delta(\Diamond\varphi) + \delta(\Diamond\diamondsuit\varphi) \\ &= \delta(\Gamma, \Diamond\varphi, \Diamond\diamondsuit\varphi) \end{aligned}$$

The case $\boxtimes$ is dual and is omitted. ■

Lemma 3.47 (Termination of Algorithm 3.42) *Let* Γ *be a history-free sequent. Then Algorithm 3.42 applied to* Γ*, (i) terminates, and (ii) yields a finite preproof.*

Proof. Part (i). We have to prove that all loops terminate. We consider the loops separately, beginning with the inner ones. There are six inner loops: lines 8–10, 11–12, 13–15, 16–17, 18–19, and 20–21.

- Lines 11–12 and 16–17 (application of foc' and respectively of $\boxtimes'_H$ as long as possible): the rule foc' can only be applied finitely many times on a finite set of sequents, since once the rule is applied to a node, it is not applicable to its child node. The same is true for the $\boxtimes'_H$ rule.
- Lines 20–21 (application of $\Box'$ as long as possible): each application of the rule $\Box'$ eliminates at least one $\Box_i$ connector. Again, since we have a finite set of sequents and sequents are finite by Definition 3.10, this process must terminate.
- Lines 8–10, 13–15, and 18–19: by König's Lemma [5, 47], it suffices to show that these loops do not yield infinite branches. Any leaf $\Sigma : l$ to which a rule $\vee'$, $\diamondsuit'$, $\wedge'$ or $\boxtimes'$ is applied yields child nodes with strictly decreasing values of δ (Fact 3.46.) When $\delta(\Sigma) = 0$, then Σ is locally reduced and none of the given rules apply.

It remains to consider the outer loop (lines 5–22). Whenever the outer loop is entered, a node is added to the preproof because there is at least one non-terminal node to which a rule is applied. Since the rules have only finitely many premises, non termination implies by König's Lemma the existence of an infinite branch. Moreover, it must be an infinite branch with infinitely many occurrences of $\Box'$ because of the termination of the inner loops. But by Lemma 3.39, such a branch must contain cyclic nodes and then the while-loop is exited; thus we cannot have an infinite branch.

Part (ii): the termination of the algorithm implies the finiteness of the preproof, since only finitely many nodes are added in each iteration. ■

Observe that all cyclic nodes in the preproof obtained by Algorithm 3.42 are unsucessful, since they have no premises and are not axiomatic.

The idea of the completeness proof is to construct a countermodel for an extended sequent $\Gamma : l(\Gamma)$ whenever $S'_{\mathrm{CK}} \nvdash \Gamma : l(\Gamma)$. The procedure to do so is given by Definition 3.49.

Notation: in the following definitions we use the notation $\mathsf{nodes}(\tau)$ to denote the set of nodes of a tree τ.

Definition 3.48 *Let $\mathcal{D}$ be the preproof in S'_{CK} resulting from a failed execution of Algorithm 3.42. The $\mathsf{twin}_{\mathcal{D}}$ function, which maps cyclic leaves of $\mathcal{D}$ into nodes of $\mathcal{D}$, is defined as follows: $\mathsf{twin}_{\mathcal{D}}(\pi_1) = \pi_0$ if and only if π_0 is the node that caused π_1 to be cyclic.*

The aim of the following definition is the extraction of a model from a preproof $\mathcal{D}$ which results from a failed execution of Algorithm 3.42. To achieve that, we transform $\mathcal{D}$ into a tree τ_2 with intermediate steps τ_0 and τ_1. In all cases, we want a notion of (1) what i-premises are in the new trees, (2) what does it mean for a leaf to be cyclic in the new trees, and (3) how does a function mapping cyclic leaves into the nodes that caused them to be cyclic look like in the new trees.

Definition 3.49 (Extraction of a model in CK) *Let $\mathcal{D}$ be the preproof resulting from a failed execution of Algorithm 3.42 with input Γ. We define the following procedure to construct a model:*

1. *Extract from* $\mathcal{D}$ *the tree* τ_0 *by dropping all priority lists. There is an obvious bijection* $\mathsf{h} : \mathsf{nodes}(\mathcal{D}) \to \mathsf{nodes}(\tau_0)$. *The notions of i-premise, cyclicity and* twin *function carry over in the natural way from* $\mathcal{D}$ *to* τ_0.

2. *Extract from* τ_0 *the tree* τ_1 *as follows: starting from the root of* τ_0 *upwards, for each instance of a rule other than* $\Box'$ *select the leftmost unsuccessful premise and drop all others. Observe that there is always an unsuccessful premise because of the assumption. Observe also that in the case of rules with one premise, nothing is dropped. The tree* τ_1 *has only unsuccessful nodes and branches out only in the instances of* $\Box'$. *There is an obvious inclusion* $\mathsf{g} : \mathsf{nodes}(\tau_1) \hookrightarrow \mathsf{nodes}(\tau_0)$. *The notions of i-premise, cyclicity and* twin *function carry over from* τ_0 *to* τ_1*: if* π *is a cyclic leaf of* τ_1*, then* $\mathsf{twin}_{\tau_1}(\pi) = \mathsf{g}^{-1}(\mathsf{twin}_{\tau_0}(\mathsf{g}(\pi)))$. *Since we prune whole subtrees, if a cyclic node* π *of* τ_0 *is in* τ_1*, then the node that caused* π *to be cyclic is also in* τ_1 *and thus in the image of* g. *Hence* g^{-1} *is defined for this latter node.*

3. *Extract the tree* τ_2 *from* τ_1 *as follows: for each branch of* τ_1*, collapse all nodes that are connected and not separated by an instance of* $\Box'$*, and label the resulting nodes with the union of the labels of the collapsed nodes. This induces a function* $\mathsf{f} : \mathsf{nodes}(\tau_1) \to \mathsf{nodes}(\tau_2)$ *whose restriction* f_ℓ *to leaves is a bijection, since there is exactly one leaf of* τ_1 *collapsed in each leaf of* τ_2*, all others having been pruned away. The notions of i-premise, cyclicity and* twin *function carry over from* τ_1 *to* τ_2*: if* $s, t \in \mathsf{nodes}(\tau_2)$*, then we define*

   - t *is an i-premise of* s *if and only if there are* $\pi_0, \pi_1 \in \mathsf{nodes}(\tau_1)$ *such that* $\mathsf{f}(\pi_0) = s$, $\mathsf{f}(\pi_1) = t$ *and* π_1 *is an i-premise of* π_0.
   - *A leaf* t *of* τ_2 *is cyclic if and only if* $\mathsf{f}_\ell^{-1}(t)$ *is cyclic. Further,* $\mathsf{twin}_{\tau_2}(t) = \mathsf{f}(\mathsf{twin}_{\tau_1}(\mathsf{f}_\ell^{-1}(t)))$.

4. *The model* $\mathcal{M} = (S, R, v)$ *is defined as follows:*

   - *The set* S *is the set of nodes of the tree* τ_2.

- $(s,t) \in R_i$ *if either* t *is an* i*-premise of* s*, or* $\mathsf{twin}_2(s) = u$ *and* t *is an* i*- premise of* u*.*
- $v(p) = \{s \mid \neg p \in s\}$.

In the following example, Definition 3.49 is applied to a failed execution of Algorithm 3.42.

Example 3.50 Let $\mathcal{A} = \{1\}$, let $\Gamma = \{(\neg p \vee \neg q), (\boxtimes p \wedge \boxtimes q)\}$ and let $l(\Gamma) = \{\boxtimes p, \boxtimes q\}$. Then Algorithm 3.42 gives the preproof of Figure 3.7, where the lists have been omitted (the active formulæ in the conclusions are underlined except in the $\Box'$ rule and in the axioms:)

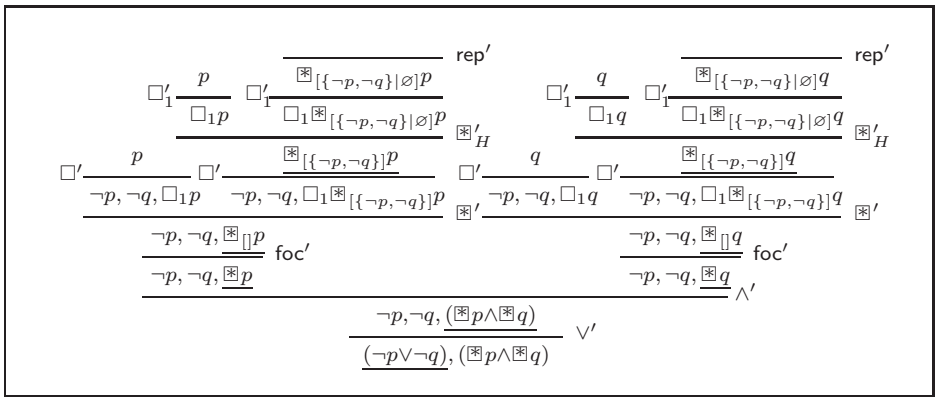


Fig. 3.7: A failed execution of Algorithm 3.42.

The tree τ_1 is obtained by choosing the leftmost failed branch for each instance of rules $\wedge'$, $\boxtimes'$ and $\boxtimes'_H$ starting from the root upwards. The resulting tree and the model extracted thereof are shown in Figure 3.8. Observe that the model is in this case a countermodel of $\Gamma : l$. This is not a mere coincidence, as we will show later on.

♠

The following example shows how cyclic nodes are dealt with.

Example 3.51 Assume that after having applied of the process of Definition 3.49 we get a tree with the shape shown in Figure 3.9. The nodes s_1

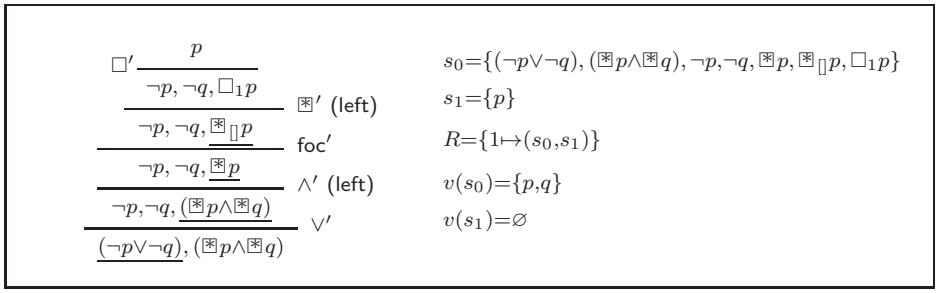


Fig. 3.8: The tree τ_1 (left) and the model (right) extracted from the failed execution of Figure 3.7.

and s_3 and the nodes s_0 and s_5 are cyclic and must have the same premises, as shown in Figure 3.10.

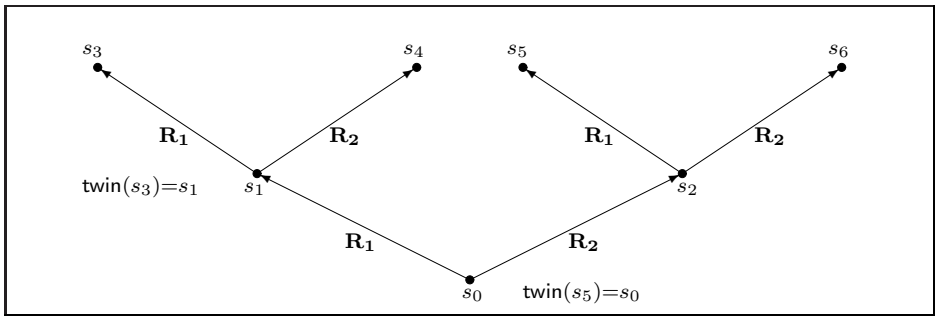


Fig. 3.9: A tree resulting from the process of Definition 3.49.

♠

Notation: in the following definition, given a tree τ, $\mathsf{parent}_\tau(\pi)$ is a partial function on $\mathsf{nodes}(\tau)$ such that $\mathsf{parent}_\tau(\pi)$ is the parent node of π. Of course, the function is undefined for the root of the tree.

Definition 3.52 (Height of a node of a tree) *Let τ be a tree and let π be a node of τ. The* height *of π, denoted by $\mathsf{height}_\tau(\pi)$, is inductively defined as follows:*

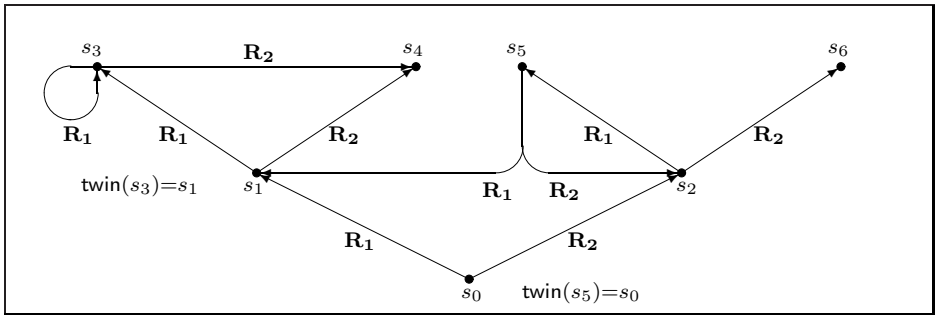


Fig. 3.10: The relations in the model derived from Figure 3.9.

$$\begin{array}{lcll} \mathsf{height}_\tau(\pi) & = & 0 & \text{if } \pi \text{ is the root of the tree;} \\ \mathsf{height}_\tau(\pi) & = & \mathsf{height}_\tau(\mathsf{parent}_\tau(\pi)) + 1 & \text{otherwise.} \end{array}$$

From now on until the end of this section, whenever we refer to a model $\mathcal{M}$ constructed according to Definition 3.49, the trees $\mathcal{D}$, τ_0, τ_1, and τ_2 will refer to the trees that are successively constructed to get the model $\mathcal{M}$.

If the tree τ is clear from the context, we just write $\mathsf{height}\,(\pi)$ and $\mathsf{parent}\,(\pi)$ instead of $\mathsf{height}_\tau(\pi)$ and $\mathsf{parent}_\tau(\pi)$ respectively.

Fact 3.53 *Let $\pi_0, \ldots, \pi_n$ be the nodes of τ_1 collapsed in a node π of τ_2 in Definition 3.49. Then, for all $i, j \in \{0, \ldots, n\}$, $i \neq j$ implies* $\mathsf{height}\,(\pi_i) \neq \mathsf{height}\,(\pi_j)$.

Proof. Immediate from the fact that the nodes $\pi_0, \ldots, \pi_n$ are all connected in τ_1 and that the only branching rule in τ_2 is $\Box'$. If a node π_i is a conclusion of $\Box'$, then its premises are collapsed in other nodes of τ_2 (see Definition 3.49.) ■

Notation: if $\mathcal{M} = (S, R, v)$ is a model obtained by application of Definition 3.49 and $s \in S$ is a node of τ_2 labelled with Γ, we write $\varphi \in s$ if $\varphi \in \Gamma$.

In the following propositions we will assume that a model $\mathcal{M} = (S, R, v)$ is constructed according to Definition 3.49 and that the nodes $\pi_1, \ldots, \pi_m$ of τ_1, respectivey labelled with $\Gamma_1, \ldots, \Gamma_m$, are collapsed in the node s of τ_2 and that they are ordered by height (this is always possible because of Fact 3.53.) We denote by $\wedge_{\text{left}}$ or $\wedge_{\text{right}}$ an instance of $\wedge$ with its right or left premise dropped. The same convention apply to the rules $\boxtimes$ and $\boxtimes_{[H]}$. We have thus the following situation:

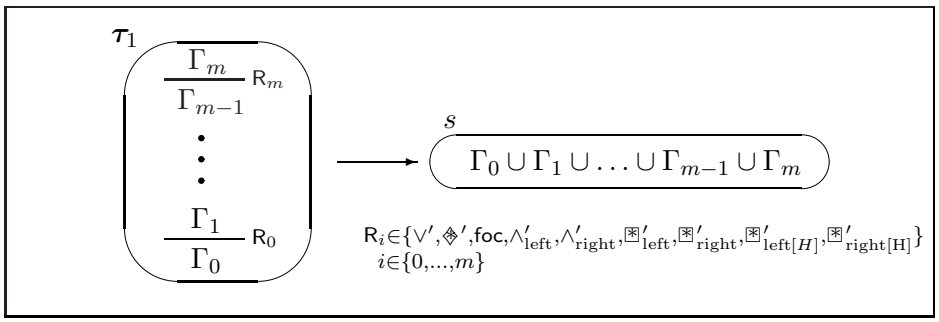


Fig. 3.11: The collapsing process from τ_1 to τ_2.

An important observation in Figure 3.11 is that Γ_m can only be a locally reduced sequent or an irreducible sequent. This is because all cyclic sequents are similar to a conclusion of a $\Box'$ rule and are thus locally reduced. In the case where Γ_m is a conclusion of a $\Box'$ rule it must be locally reduced. Since no other rule than $\Box'$ must be applicable to Γ_m, since otherwise there would be another node labelled Γ_{m+1} in τ_1 collapsed in s with $\mathsf{height}(\Gamma_{m+1}) > \mathsf{height}(\Gamma_m)$, and π_m cannot be axiomatic, the only other possibility is that it be irreducible.

Proposition 3.54 *Let $\mathcal{M} = (S, R, v)$ be a model constructed according to Definition 3.49 and let $s \in S$. Then*

(i) *If $\boxtimes_{[H]}\varphi \in s$, then $\Box\varphi \in s$ or for some H', $\Box\boxtimes_{[H']}\varphi \in s$.*

(ii) *If $\boxtimes\varphi \in s$, then $\Box\varphi \in s$ or $\Box\boxtimes\varphi \in S$ or for some H', $\Box\boxtimes_{[H']}\varphi \in s$.*

Proof. Part (i). If $s \in S$, then s is a node of τ_2 containing the collapsed nodes $\pi_0, \ldots, \pi_m$ of τ_1. Recall that the nodes $\pi_0, \ldots, \pi_m$ are ordered by height and labelled with $\Gamma_0, \ldots, \Gamma_m$ and thus Γ_m is the label of the highest node of τ_1 collapsed in s. Since $\boxtimes_{[H]}\varphi \in s$, then $\boxtimes_{[H]}\varphi \in \Gamma_i$ for some i, $1 \leq i \leq m$.

Assume that the statement does not hold, i.e., that $\boxtimes_{[H]}\varphi \in s$ but there is no $k \in \{0, \ldots, m\}$ such that $\Box\varphi \in \Gamma_k$ or $\Box\boxtimes_{[H']}\varphi \in \Gamma_k$. Thus, the rule $\boxtimes'_H$ was not applied to any node π_j of τ_1, $m \geq j \geq i$. Hence, $\boxtimes_{[H]}\varphi \in \Gamma_j$ for all j such that $m \geq j \geq i$ and in particular $\boxtimes_{[H]}\varphi \in \Gamma_m$. Therefore, Γ_m is neither an irreducible sequent (the rule $\boxtimes'_{[H]}$ is applicable to it) nor a locally reduced sequent, which yields a contradiction.

Part (ii). Assume $\boxtimes\varphi \in s$, and $\Box\varphi \notin s$, $\Box\boxtimes\varphi \notin s$ and $\Box\boxtimes_{[H]}\varphi \notin s$ then the rule $\boxtimes$ was not applied to any node π_j of τ_1, $m \geq j \geq i$. Because of part (i) of this Proposition, the formula was also not annotated. Thus, by an analogous reasoning as part (i), $\boxtimes\varphi \in \Gamma_m$ and we get a contradiction. ■

Fact 3.55 *Let $\mathcal{M} = (S, R, v)$ be a model obtained by application of Definition 3.49. Then $\mathcal{M}$ is finite.*

Proposition 3.56 *Let $\mathcal{M} = (S, R, v)$ be a model obtained by application of Definition 3.49 and let $s \in S$. If $\Box_i\boxtimes_{[H]}\varphi \in s$, then there is an H' and a $t \in S$ such that $\boxtimes_{[H']}\varphi \in t$ and $(s, t) \in R_i$.*

Proof. As before, we assume that s is a node of τ_2 containing the collapsed nodes $\pi_0, \ldots, \pi_m$ of τ_1 ordered by height and labelled with $\Gamma_0, \ldots, \Gamma_m$. Then $\Box_i\boxtimes_{[H]}\varphi \in \Gamma_m$, since the only rule that can process this formula is $\Box'$. Therefore, the node π_m is not irreducible.

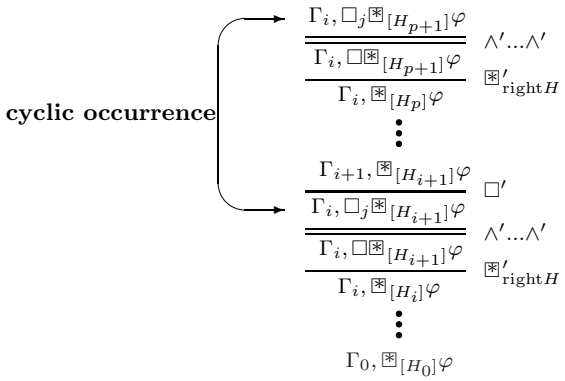
If the node π_m is not cyclic, then there is a node π_{m+1} collapsed in a state t which is an i-premise of π_m and contains $\boxtimes_{[H]}\varphi$, and $(s, t) \in R_i$.

If π_m is cyclic, there is a non-cyclic node $\mathsf{twin}(\pi)$ labelled Σ in τ_1 such that $\Sigma \sim \Gamma_m$ and thus $\Box_i\boxtimes_{[H']}\varphi \in \Sigma$. Since the node is not cyclic, it has an i-premise π' labelled Σ' that is collapsed in a state t containing $\boxtimes_{[H']}\varphi$ and $(s, t) \in R_i$ by the construction of τ_2. ■

Proposition 3.57 *Let $\mathcal{M} = (S, R, v)$ be a model constructed according to Definition 3.49 and let $s \in S$. If $\boxtimes_{[H]}\varphi \in s$, then for all s-paths $s_0, s_1, \ldots$ there is a $k < \omega$ such that $\Box\varphi \in s_k$.*

Proof. Assume that the statement does not hold for some state $s \in S$ with $\boxtimes_{[H_0]}\varphi \in s$. Thus there must be some s-path $s_0, s_1, \ldots$ such that for all s_j in it, $\Box\varphi \notin s_j$.

By assumption and Proposition 3.54, $\boxtimes_{[H_0]}\varphi \in s_0$ implies $\Box\boxtimes_{[H_1]}\varphi \in s_0$ and by Proposition 3.56, $\boxtimes_{[H_1]}\varphi \in s_1$. By a straightforward induction, we have that for all j in the path, $\boxtimes_{[H_j]}\varphi \in s_j$ and $\Box\boxtimes_{[H_{j+1}]}\varphi \in s_j$. Thus all states s_j in the path have a successor and the path is infinite. By Fact 3.55, an infinite path has a loop. This means that there is a branch that has the following shape in the tree τ_1. Double bars symbolise the successive applications of rule $\wedge'$ to get $\Box_i\boxtimes_{[H]}\varphi$ starting from $\Box\boxtimes_{[H]}\varphi$ (steps 18–19 of Algorithm 3.42.)



The nodes respectively labelled with Γ_i, $\Box_j\boxtimes_{[H_{i+1}]}\varphi$ and Γ_i, $\Box_j\boxtimes_{[H_{p+1}]}\varphi$ result from applications of $\wedge'$ on right premises of rule $\boxtimes'_H$ and there is no cyclic node between them. Thus we have

$$\Gamma_i \in H_{i+1} \subset H_p$$

Therefore, the node labelled with Γ_i, $\boxtimes_{[H_p]}\varphi$ is actually an instance of rep',

which is not allowed by construction and we get thus a contradiction. ■

Observe that as a consequence of Proposition 3.57, any formula that is annotated must eventually be unannotated in the course of a failed proof-search. The following Lemma shows that the model obtained by Definition 3.49 is indeed a countermodel of Γ.

Lemma 3.58 (Definition 3.49 yields a countermodel) *Let Γ be a history-free sequent such that Algorithm 3.42 applied to it fails (see Definition 3.43) yielding a preproof $\mathcal{D}$ and let $\mathcal{M} = (S, R, v)$ be the model constructed according to Definition 3.49. Then, if φ is a formula, $s \in S$ and $\varphi \in s$, then $(\mathcal{M}, s) \models \neg\varphi$.*

Proof. Induction on the structure of φ.

Base case: $\varphi = p$ or $\varphi = \neg p$. The result follows immediately from Definition 3.49: if $\neg p \in s$, then $p \in v(s)$ and thus $(\mathcal{M}, s) \models p = \neg\neg p$. If $p \in s$, then $\neg p \notin s$ (by construction the nodes cannot be axiomatic) and thus $p \notin v(s)$. Therefore, $(\mathcal{M}, s) \not\models p$ and hence $(\mathcal{M}, s) \models \neg p$.

Induction step:

If $\varphi = (\alpha \wedge \beta) \in s$ then by the $\wedge'$ rule $\alpha \in s$ or $\beta \in s$ and by induction hypothesis either $(\mathcal{M}, s) \models \neg\alpha$ or $(\mathcal{M}, s) \models \neg\beta$. Thus $(\mathcal{M}, s) \models (\neg\alpha \vee \neg\beta) = \neg(\alpha \wedge \beta)$.

If $\varphi = (\alpha \vee \beta) \in s$ then by the $\vee'$ rule $\alpha, \beta \in s$ and by induction hypothesis $(\mathcal{M}, s) \models \neg\alpha$ and $(\mathcal{M}, s) \models \neg\beta$. Thus $(\mathcal{M}, s) \models (\neg\alpha \wedge \neg\beta) = \neg(\alpha \vee \beta)$.

If $\varphi = \Box_i\alpha \in s$ then there are two possibilities:

- The state s is not cyclic. Thus by the $\Box'$ rule there is some t which is an i-premise of s such that $\alpha \in t$. In this case, $(s, t) \in R_i$ by construction and by induction hypothesis $(\mathcal{M}, t) \models \neg\varphi$ and thus $(\mathcal{M}, s) \models \neg\Diamond_i\neg\varphi = \neg\Box_i\varphi$.

- The state s is cyclic and there is a state $u \in S$ such that $u = \mathsf{twin}(s)$ and $\Box_i\alpha \in u$. Thus by the $\Box'$ rule u has an i-premise t with $\alpha \in t$.

In this case, $(s,t) \in R_i$ by construction and by induction hypothesis $(\mathcal{M},t) \models \neg\varphi$ and thus $(\mathcal{M},s) \models \neg\Diamond_i\neg\varphi = \neg\Box_i\varphi$.

If $\varphi = \Diamond_i\alpha \in s$, then either in all states t that are i-premises of s, $\alpha \in t$ or there is a state $u \in S$ such that $u = \mathsf{twin}(s)$ and for all i-premises t of u, $\alpha \in t$. Since in both cases $(s,t) \in R_i$, and by induction hypothesis $(\mathcal{M},t) \models \neg\alpha$ for all the states t, we have that $(\mathcal{M},s) \models \Box_i\neg\alpha = \neg\Diamond_i\alpha$.

If $\varphi = \Diamond\!\!\!\!-\alpha \in s$, we prove that $\Diamond\alpha \in t$ and $\Diamond\Diamond\!\!\!\!-\alpha \in t$ for all states t that are reachable from s. It suffices to show that the statement holds for all states that are reachable from s in m steps for some m. The proof proceeds by induction on m. The base case ($m = 0$ and thus $t = s$) is immediate. Assume that the statement holds for all states u that are reachable from s in q steps. Since $\Diamond\Diamond\!\!\!\!-\alpha \in u$, for all states v with $(u,v) \in R_{\mathcal{A}}$, it is the case that $\Diamond\!\!\!\!-\alpha \in v$. Thus, $\Diamond\alpha \in v$ and $\Diamond\Diamond\!\!\!\!-\alpha \in v$. Hence, for all states $t \in S$ such that $(s,t) \in R^+_{\mathcal{A}}$, $(\mathcal{M},t) \models \neg\alpha$ and thus for all states $t \in S$ such that $(s,t) \in R^*_{\mathcal{A}}$, $(\mathcal{M},t) \models \neg\Diamond\alpha$. Thus, $(\mathcal{M},s) \models \boxdot\neg\alpha = \neg\Diamond\!\!\!\!-\alpha$.

If $\varphi = \boxdot\alpha \in s$, we prove that for all s-paths $s_0, \ldots$ there is a $k < \omega$ such that $\Box\alpha \in s_k$. If this is not the case, then there is an s-path in which the right branch has been kept in all instances of rule $\boxdot$ or rule $\boxdot_H$ respectively applied to $\boxdot\alpha$ or to $\boxdot_{[H]}\alpha$ when the tree τ_1 was constructed (see Definition 3.49.) Since for all states s_j of the path it is the case that $\boxdot\alpha \in s_j$ or $\boxdot_{[H]}\alpha \in s_j$, then by Proposition 3.54 it is also the case that $\Box\boxdot\alpha \in s_j$ or $\Box\boxdot_{[H]}\alpha \in s_j$ and thus there is always a successor of s_j. Thus, such a path should be infinite. The result follows by Proposition 3.57: all formulæ of the form $\boxdot\psi$ in an infinite path must eventually be annotated and unannotated; thus there is a state s_q in which $\boxdot\alpha$ is annotated, and by Proposition 3.57, there is a state s_p which is reachable from s_q such that $\Box\alpha \in s_p$ and by induction hypothesis, $(\mathcal{M},s_p) \models \neg\Box\alpha$ and thus there is a state t which reachable from s_p in one step with $(\mathcal{M},t) \models \neg\alpha$. Since s_q is reachable from $s_0 = s$, so are s_p and t. Hence $(\mathcal{M},s) \models \neg\boxdot\alpha$. ■

Corollary 3.59 *There is a decision procedure for establishing the validity of a formula φ in S'_{CK}.*

Proof. Given a formula φ, it is always possible to apply Algorithm 3.42 to construct a preproof. Since the resulting preproof is finite by Lemma 3.47, it is possible to decide whether the execution has been successful or not and thus whether the formula is valid or not. ■

Lemma 3.60 (Completeness of S'_{CK}) *Let Γ be a history-free sequent and let $l(\Gamma)$ be a list of all formulæ of the form $\boxdot\varphi$ that are in $\mathsf{cl}(\Gamma)$. Then $\models \Gamma$ implies $S'_{CK} \vdash \Gamma : l(\Gamma)$.*

Proof. Immediate from Lemma 3.58. If $S'_{\mathrm{CK}} \nvdash \Gamma : l(\Gamma)$, then execution of Algorithm 3.42 on Γ fails and it is possible to construct a countermodel for $\Gamma : l(\Gamma)$ according to Definition 3.49. ■

Theorem 3.61 (Completeness of S_{CK}) *The system S_{CK} is complete for history-free sequents.*

Proof. Direct from Lemmas 3.31 and 3.60. By Lemma 3.31, any proof in S'_{CK} can be transformed in a proof in S_{CK}. The completeness of S_{CK} follows from that of S'_{CK} (Lemma 3.60.) ■

3.6 Complexity

The goal of this section is to determine an upper bound for the number of nodes that the tree (strictly speaking a loop-tree [78]) constructed with Algorithm 3.42 may have. We follow the notation of [70].

We state first some auxiliary results before we begin. We will use the notation $\mathsf{succ}(\pi)$ for some sucessor of a node π in a tree τ. First we recall some well-known facts about trees.

Definition 3.62 (Degree and height of a tree) *Let τ be a tree and let π be a node of τ. The* degree *of π is the number of child nodes it has. The* degree *of τ, denoted by $\mathsf{deg}(\tau)$ is defined as*

$$\mathsf{deg}(\tau) = \mathsf{max}\{\mathsf{deg}(\pi) \mid \pi \text{ is a node of } \tau\}$$

The height *of τ, denoted by* $\mathsf{height}(\tau)$ *is defined as*

$$\mathsf{height}(\tau) = \mathsf{max}\{\mathsf{height}(\pi) \mid \pi \text{ is s node of } \tau\}$$

Proposition 3.63 (Number of leaves in a tree) *Let τ be a tree with* $\mathsf{height}(\tau) > 0$. *Then the number of leaves of τ is bounded by* $\mathsf{deg}(\tau)^{\mathsf{height}(\tau)}$.

Proof. Induction on $\mathsf{height}(\tau)$.

Base case: if $\mathsf{height}(\tau) = 1$, then the tree has $\mathsf{deg}(\pi_0)$ leaves, where π_0 is the root, and also $\mathsf{deg}(\pi_0) = \mathsf{deg}(\tau) = \mathsf{deg}(\tau)^{\mathsf{height}(\tau)}$.

Induction step: if $\mathsf{height}(\tau) = q + 1$, we know by induction hypothesis that there are $\mathsf{deg}(\tau)^q$ nodes at height q. Since each one of them may have up to $\mathsf{deg}(\tau)$ child nodes, we get that the number of nodes at height $(q+1)$ is at most

$$\mathsf{deg}(\tau)^q * \mathsf{deg}(\tau) = \mathsf{deg}(\tau)^{q+1}$$

■

Lemma 3.64 (Number of nodes in a tree) *Let τ be a tree. Then the number of nodes of the tree is in* $\mathsf{deg}(\tau)^{\mathcal{O}(\mathsf{height}(\tau))}$.

Proof. By Proposition 3.63 the number of leaves at height h is at most $\mathsf{deg}(\tau)^{\mathsf{height}(\tau)}$, if we denote the number of nodes by n_n, the degree of the tree by q, and the height of the tree by h, we have:

$$n_n = q^h + q^{h-1} + q^{h-2} + \ldots + q^1 + 1 < \underbrace{q^h + \ldots + q^h}_{h \text{ times}} + 1$$

Now we show that $h * q^h + 1$ is in $q^{\mathcal{O}(h)}$.

$$\begin{aligned} h * q^h + 1 &= 2^{\mathsf{log}_2 h} * 2^{h * \mathsf{log}_2 q} + 1 = 2^{\mathsf{log}_2 h + h * \mathsf{log}_2 q} + 1 \\ &< 2^{h * \mathsf{log}_2 q + h * \mathsf{log}_2 q} + 1 = 2^{2 * h * \mathsf{log}_2 q} + 1 \\ &\subseteq 2^{\mathcal{O}(h * \mathsf{log}_2 q)} = 2^{\mathcal{O}(h) * \mathsf{log}_2 q} = q^{\mathcal{O}(h)} \end{aligned}$$

■

In the case of preproofs in S'_{CK}, the degree is determined by rule $\Box'$, since in all other cases we have that the rules have at most two premises. The rule $\Box'$ has as many premises as formulæ of the form $\Box_i\psi$ appear in the conclusion.

Fact 3.65 *Let Γ be a sequent. Then the size of $\mathsf{cl}(\Gamma)$ is in $\mathcal{O}(\mathsf{size}(\Gamma))$.*

Proof. Immediate from Lemma 3.36. The size of a closure is bounded by $\delta_{\mathcal{A}}(\Gamma)$ (see Definition 3.34), which is at most $4 * |\mathcal{A}|$ times $\mathsf{size}(\Gamma)$. Thus, $\delta_{\mathcal{A}}(\Gamma)$ is in $\mathcal{O}(\mathsf{size}(\Gamma)$. ■

To determine how many nodes there are in the tree constructed with Algorithm 3.42 we begin by determining the length of a branch.

Lemma 3.66 (Length of a branch in a preproof in S'_{CK}) *Let $\mathcal{D}$ be the tree obtained by Algorithm 3.42 applied on a sequent Γ and let $n = \mathsf{size}(\Gamma)$. Then in the worst case, the number of nodes in a branch is in*

$$\mathcal{O}(n! * 2^n).$$

Proof. A branch is closed only when an axiomatic node, an irreducible node or a cyclic node is reached. If a repetition of a sequent $\Gamma : l$ is reached without any instance of the foc' rule in between, the cyclic node is reached in the worst case after $\mathcal{O}(2^n)$ nodes. But if shortly before reaching the cyclic node an instance of foc' occurs, and the same situation repeats itself for all possible permutations of the list then the worst case is that all possible permutations of the list must be processed before the repetition occurs. The number of possible lists (again in the worst case) is in $\mathcal{O}(n!)$. Therefore, in the worst case the name of nodes in a branch is in $\mathcal{O}(n! * 2^n)$. ■

Fact 3.67 (Complexity of Algorithm 3.42) *Let $n = \mathsf{size}(\Gamma)$. Then the worst-case complexity of Algorithm 3.42 applied on a sequent Γ is in*

$$2^{\mathcal{O}((n+1)!*2^n)}.$$

Proof. The degree of the tree is in $\mathcal{O}(n)$, since it is determined by the rule $\Box'$. If we denote by $n_{\mathcal{D}}$ the total number of nodes, and taking into account that by Lemma 3.66 the height of the tree is in $\mathcal{O}(n! * 2^n)$, we have by Fact 3.64:

$$\begin{aligned} n_{\mathcal{D}} &\subseteq \mathcal{O}(n)^{\mathcal{O}(n!*2^n)} && (3.34)\\ &= 2^{\mathsf{log}_2\mathcal{O}(n)*\mathcal{O}(n!*2^n)} && (3.35)\\ &\subset 2^{\mathcal{O}(n)*\mathcal{O}(n!*2^n)} && (3.36)\\ &\subset 2^{\mathcal{O}((n+1)!*2^n)} && (3.37) \end{aligned}$$

■

As a consequence of Lemmas 3.64 and 3.66, we have that worst case complexity is even worse than double exponential, as in [1], which is already intractable.

The worst case occurs when there are formulæ of the form $\boxtimes\psi$ making intermittent appearances. Nevertheless, we cannot ignore the lists when considering repetitions. The following example shows that a sequent could be repeated with different lists.

Example 3.68 Assume for simplicity $\mathcal{A} = \{1\}$. We make a preproof of the extended sequent

$$\Gamma = \{\Box p, \Box q, \diamondsuit(\boxtimes p \wedge p), \diamondsuit(\boxtimes q \wedge q), \diamondsuit\Box p\} : [\boxtimes p, \boxtimes q]$$

We write only the branch of the preproof in which the repetition of the sequent with reversed lists occurs. Besides, we group several rules together. The underlined formulæ in the conclusions are the active ones. We use the following abbreviations:

$$\begin{aligned} \varphi &:= \diamondsuit(\boxtimes p \wedge p)\\ \psi &:= \diamondsuit(\boxtimes q \wedge q)\\ \zeta &:= \diamondsuit\Box p \end{aligned}$$

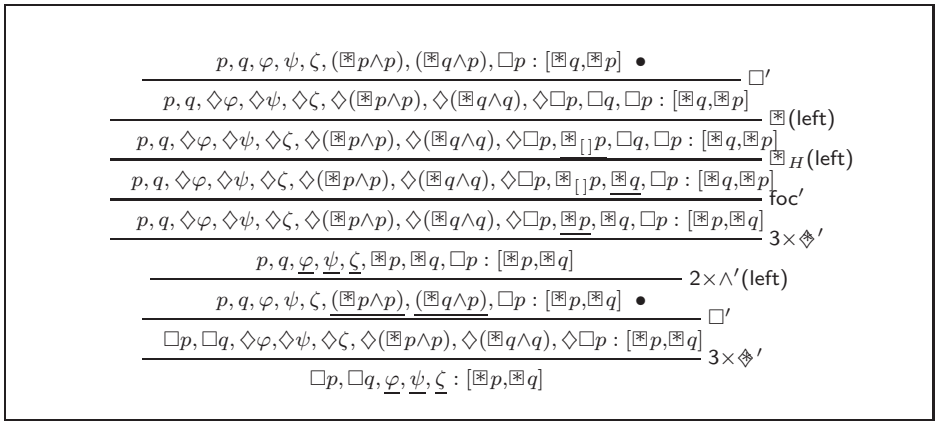


Fig. 3.12: A repetition of a sequent with different priority lists.

As shown in Figure 3.12, both sequents and lists must be taken into account for the repetitions. ♠

3.7 Conclusions

The method has some nice properties: it is cut-free, it has potential for parallelisation, since branching rules give rise to independent subtrees. These subtrees could be treated as separate processes [8] or as separate threads [81]. Besides, there is no ω-rules as in [2, 14]. One drawback is that there is no known syntactic cut-elimination for this method. But the main drawback is the complexity of the method in the worst case, as we have seen in Section 3.6. The complexity of the decision problem in CK is known to be EXPTIME-complete [37]. Worst case here is much worse.

Notwithstanding that, there are some nice points in the method that make an implementation interesting. It is not necessary to construct the whole tree, as for instance in [27]. Thus, the worst case is not necessarily met. Besides, the other implementation we know of [1], which is based on

tableaux methods [35, 78], has a worst case which is double exponential: although much better, still intractable.

By Corollary 3.59, Definition 3.49 provides a decision procedure. This is what we have implemented. The implementation is described in the next chapter. It is in SWI-Prolog and is a quite straightforward codification of Definition 3.49.

Since the aforementioned process includes a check for weakly cyclic statements, the Prolog program implements something which goes beyond a mere proof system. Nevertheless, the implementation is quite concise and brief.

Chapter 4

Notes on the Implementation

> Double, double, toil and trouble
>
> Shakespeare, *Macbeth, Act IV, Scene I*

4.1 Introduction

In this chapter we describe some details of the implementation. We consider the internal representations of formulæ and sequents and the conversions between them, the process for the construction of a proof in S_{CK} after having obtained one in S'_{CK} (recall that this is always possible by Lemma 3.31), and the process of construction of a countermodel when the proof search process fails. The chapter closes with a description of the implementation of the proof system S'_{CK}. The whole implementation comprises 530 lines of source code and 108 defined predicates.

The chapter is organised as follows: section 4.2 gives an overview of Prolog with some considerations about the important issue of cuts. Section 4.3 explains the usage of the program and Section 4.4 describes the various data representations that are used in the implementation. Section 4.5 explains the conversions between the different normal forms and the parsing

procedure. The process of deriving a proof in S_{CK} from a proof in S'_{CK} is shown in Section 4.6. The —rather more complex— process of constructing a countermodel is briefly explained in Section 4.7. The implementation of the rules of the proof system S'_{CK} is described in Section 4.8 and the implementation of the decision procedure in Section 4.9.

4.2 A Very Brief Introduction to Prolog

This section provides a brief and informal introduction to Prolog[1], the language used for the implementation. This is intended as a general description and we will paint with very broad brush in this introduction. For more detailed information, the reader is referred to the specific literature. See for instance —among many others— [12, 21, 22, 25, 26, 49, 60, 61, 77, 80].

Prolog is a declarative high-level language. A Prolog program is a set of *definite program clauses*, or *clauses* for short, and *goals*. Clauses are implications of the form $a \Leftarrow b_1 \wedge b_2 \wedge \ldots \wedge b_q$, and goals are implications with an empty consequent. A *Horn clause* is either a definite program clause or a goal.

The set of clauses can be viewed as a database with a description of a problem and a goal as an enquiry to the database. The execution of a Prolog program is triggered by the attempt to prove (or disprove) a goal. The underlying machinery is the *resolution principle* [76].

The first implementation of Prolog in the mid-seventies [23] was based in the observation of Kowalski [51, 52] that a Horn clause like $p \Leftarrow a \wedge b \wedge c$ can be interpreted not only declaratively ("a, b and c imply p"), but also procedurally ("to solve p, solve a, then b, and then c.") Such a clause would be written in SWI-Prolog [85], a Prolog version that complies with the *de facto* standard of the language [21, 22, 61], as follows:

```
p :- a,b,c.
```

A goal like $\Leftarrow p$ is represented as

```
? -p.
```

[1] The name is an acronym for "PROgrammation en LOGique."

We will use henceforth the SWI-Prolog notation, which is the version in which the implementation was written. The only data structures in Prolog are *terms*, defined next.

Definition 4.1 (Terms in Prolog) *Terms in Prolog are either:*

- Constants *(string beginning with lowercase letters),* variables *(strings beginning with uppercase letters) or numbers;*
- *Compound terms, which consist of a functor of arity n followed by n terms called* arguments*.*

It is well-known that terms are trees. Consider for instance, the following term [80]:

```
sentence(nphrase(dogs),vbphrase(verb(like),nphrase(cheese)))
```

where sentence is the main functor (arity 2) and both of its arguments, nphrase(dogs) and vbphrase(verb(like),nphrase(chesse)), are subterms of the main term. This term is depicted in Figure 4.1.

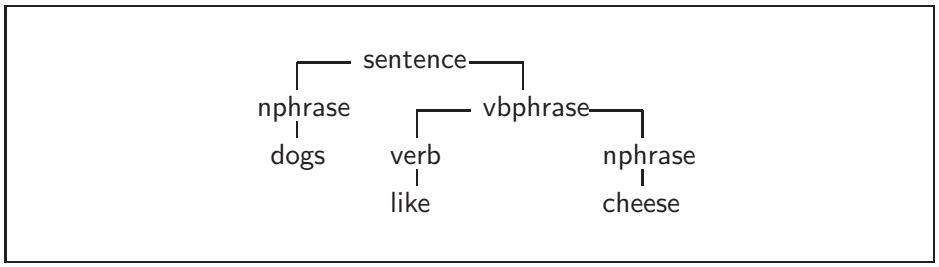


Fig. 4.1: A representation of a term as a tree.

A special type of compound terms are the *lists*. A list is a compound term represented with

- an atom [] representing the empty list, and

- a compound term with functor "." of arity 2. The arguments represent the head and the tail of the list, which is itself a list.

Lists may contain elements of any type. For instance the list containing the numbers 7, 2, and 3 (in this order) is represented as

.(7,.(2,.(3,[])))

The tree representation of the above list is the following one:

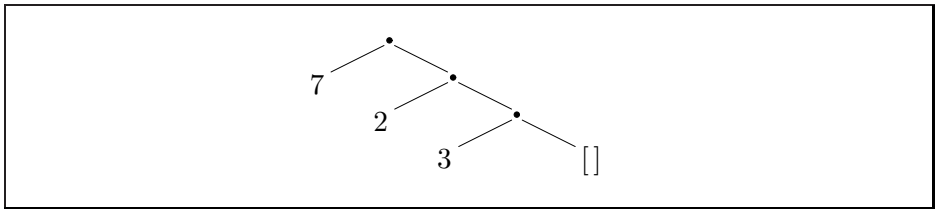


Fig. 4.2: A representation of the list [7,2,3] as a tree.

The notation .(7,.(2,.(3,[]))) for a list is rather unwieldy and not used. The usual notation for this list is [7, 2, 3]. which is the one we use henceforth.

To apply resolution the concept of *unification* is of central importance. Given two terms t1 and t2, a *unifier* is a substitution γ for the variables in t1 and t2 such that $t1\gamma = t2\gamma$.

Example 4.2 Consider the terms

$$\begin{aligned} t1 &= f(g(b),h(a,W)) \\ t2 &= f(X,h(Y,u(Z))) \end{aligned}$$

The substitution $\gamma = \{[W|u(Z)], [X|g(b)], [Y|a]\}$ unifies t1 and t2, since

$$f(g(b),h(a,W))\gamma = f(g(b),h(a,u(Z))) = f(X,h(Y,u(Z)))\gamma$$

♠

Observe that in general there is not a unique unifier for two terms. For instance the unifier $\gamma_1 = \{[\mathsf{W}|\mathsf{u}(\mathsf{p})], [\mathsf{X}|\mathsf{g}(\mathsf{b})], [\mathsf{Y}|\mathsf{a}]\}$ would also do the job. This is, though, a less general unifier than γ, since it instantiates the variable Z, which is not necessary. In general, a unifier β is less general than a unifier α if there is a substitution δ such that $\alpha\delta = \beta$. There are well-known algorithms to determine the *most general unifier* (mgu) of two terms [60].

Roughly speaking, the resolution method applied to Prolog programs works as follows: [80] we start with a program P and a goal g, and an empty substitution φ. A *resolvent* contains the terms that must be resolved; initially it contains only the goal. If there is a term t and a clause h :- $\mathsf{c}_1,\ldots,\mathsf{c}_m$ such that t and c may be unified with mgu α, then the term t in the resolvent is replaced by the terms $\mathsf{c}_1, \ldots,\mathsf{c}_m$ and φ is replaced with $\varphi\alpha$ with some suitable renaming of variables. The process goes on until the resolvent is empty, (success) or there is no head of a clause that unifies with any term of the resolvent. Consider the following example.

Example 4.3 Let us consider the following program, where the numbers at the lefthand side are only for explanatory purposes.

1. p(X) :- a(X), b(X), c(X).
2. a(1).
3. a(2).
4. a(3).
5. b(2).
6. b(3).
7. c(3).

The goal p(X) succeeds with the unification X = 3. This result is obtained as follows:

1. We begin with the resolvent

$$\rho_1 = (\mathsf{p}(\mathsf{X})).$$

2. By clause 1, the only one whose head unifies with the term of the

resolvent, we get a new resolvent

$$\rho_2 = (\mathsf{a(X)}, \mathsf{b(X)}, \mathsf{c(X)}).$$

3. Clause 2 unifies with the first term of ρ_2 with mgu=[X|1]. The new resolvent is
$$\rho_3 = (\mathsf{b(1)}, \mathsf{c(1)}).$$

4. Since no clause unifies with the first term of ρ_3, it is necessary to go back to ρ_2 and try another clause.

5. Clause 3 unifies with the first term of ρ_2 with mgu=[X|2]. The new resolvent is
$$\rho_4 = (\mathsf{b(2)}, \mathsf{c(2)}).$$

6. Clause 5 unifies with the first term of ρ_2. The new resolvent is
$$\rho_5 = (\mathsf{c(2)}).$$

7. Since no clause unifies with the first term of ρ_5, it is necessary to backtrack to ρ_4 and try another clause. Since no clause other than 5 unifies with the first term of ρ_4, it is necessary to backtrack to ρ_2.

8. Clause 4 unifies with the first term of ρ_2 with mgu=[X|3]. The new resolvent is
$$\rho_6 = (\mathsf{b(3)}, \mathsf{c(3)}).$$

9. Clause 6 unifies with the first term of ρ_6. The new resolvent is
$$\rho_7 = (\mathsf{c(3)}).$$

10. Clause 7 unifies with the first term of ρ_6. The new resolvent is $\rho_8 = ()$ and this is thus a success node.

♠

The process may be depicted in a graph in a standard way as follows: nodes are resolvents, downward edges are labelled with the clauses applied (and possibly with the unifiers) and upward edges represent backtracking. In the last example we begin with the node corresponding to the goal ρ_1. From this node, we get an arc going to ρ_2 and from the latter an arc going to ρ_3 From the last node we have to backtrack to ρ_2 and get another arc to ρ_4 and so on.

The graph corresponding to the complete execution of this example is shown in Figure 4.3.

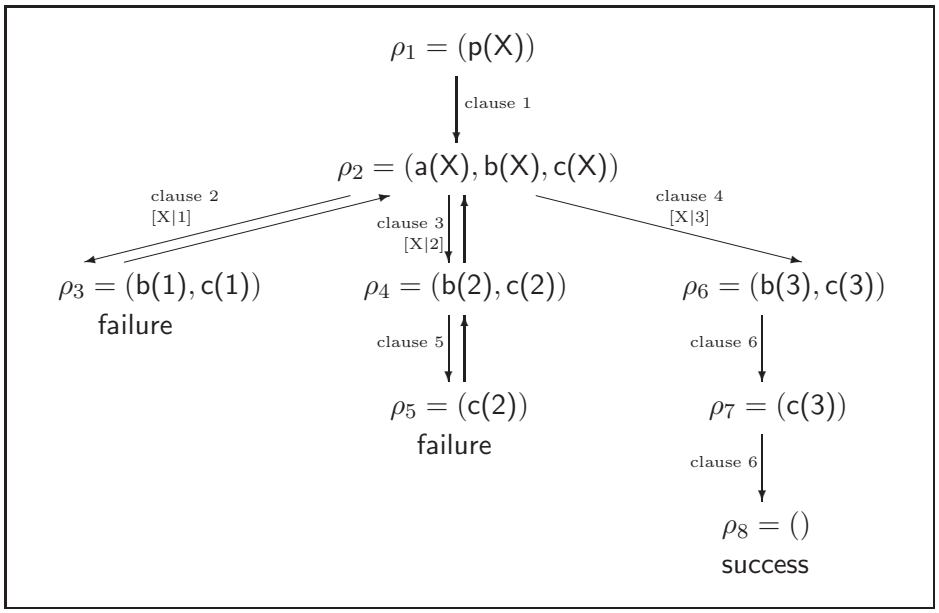


Fig. 4.3: The execution of the program of Example 4.2.

The following example shows a classical Prolog program to determine whether an element is a member of a list. This predicate is so common that most Prolog implementation have it inbuilt.

```
member(X,[H|T])  :-  X=H.
member(X,[H|T])  :-  member(X,T).
```

The two clauses form the predicate member/2 (this is standard Prolog notation; it means that the predicate member that has two parameters. The first one is the element whose membership we want to test and the second one is the list.

The predicate contains a straightforward description of the problem: an element is a member of a list if either it is the head of the list or it is a member of its tail. Observe also that there is a recursive call in the second clause, which is another distinctive feature of Prolog.

This program may be run with a goal. For instance, two possible runs of the program are the following ones:

```
?- member(1,[2,3,d,1]).
true.

?- member(w,[2,3,d,1]).
false.
```

The execution trees for these runs are shown in Figures 4.4 and 4.5. We begin with the first one, that shows the resolvents that occur in the execution of a successful goal.

Although it is usually the case that the execution ends once a successful branch has been found, it is possible to force backtracking so that all possible solutions be found. We d not use this possibility in our implementation.

The second graph shows the execution of an unsuccessful goal. The execution fails after all possibilibies have been tested and have failed.

Observe that it is always the case that the first attempt is at the first clause of the predicate member/2. After failure, it tries the recursive clause (the second one.) The process repeats itsel until the first clause succeeds.

In the second case, the recursive call is repeated until the resolvent has the term member(w,[]). Two terms whose main functors are different are not unifiable. In particular, two constants (in this case, w and []) are not unifiable and the execution returns false.

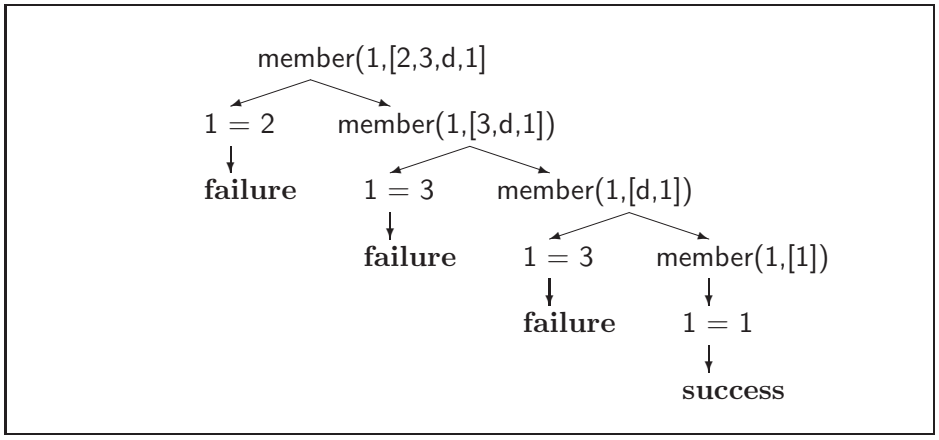


Fig. 4.4: A successful execution.

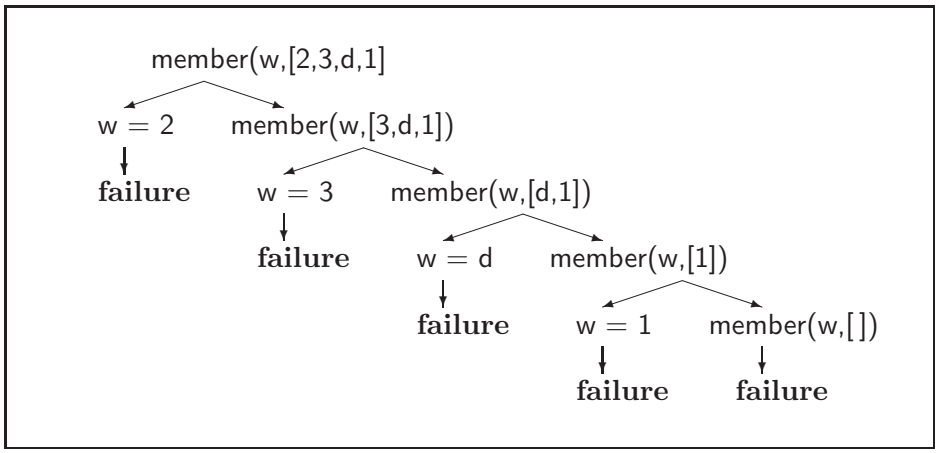


Fig. 4.5: A failed execution.

Let us cast a second glance at our program:

```
member(X,[H|T])  :-  X=H.
member(X,[H|T])  :-  member(X,T).
```

Observe that in the first clause above the variable T plays no rôle. The same is true for the variable H in the second clause. Besides, it is not necessary to use two different names of variables and then to state their equality; it is enough to use the same name for both in the head of the clause.

The following syntax is thus the usual one:

```
member(H,[H|_]).
member(X,[_|T])   :-  member(X,T).
```

We finish this section mentioning that pure logical programming is not enough for practical programming. There are thus *extralogical features* such as i/o predicates, and *meta-logical features*, which we do not use here. More details can be found in the bibliography.

As pointed out in [80], the main features that imply a departure from pure resolution are *negation as failure* (NAF), introduced in [20], and *cuts*. NAF means that the negation of a goal is true if the goal finitely fails.

It is a well-known feature of NAF that it is a nonmonotonic feature which is different from classical negation. The difference is stressed in modern implementations of Prolog by using \+ rather than not. Consider for instance the following set of clauses.

```
composer(beethoven).
composer(mozart).
```

The following goal succeeds:

```
\+composer(stravinsky).
```

This is because the predicate composer(stravinsky) cannot be proved with this set of clauses. The nonmonotonic nature of this feature is clear: if the clause composer(stravinsky) is added to the set of clauses, the goal is no longer successful.

NAF tends to be confused with CWA, which is actually stronger: in CWA, something is taken to be false in a program if it is not a consequence

of it; in NAF something is taken to be false if it the attempts to prove it *finitely* fail. A goal leading to an infinite branch would be false from the point of view of CWA and not from that of NAF. From a practical point, implementing anything beyond NAF is hard. In our implementation negation as failure does not occur.

Cuts, denoted by an exclamation mark (!), are a controversial feature. On the one hand, they can greatly increase the efficiency of Prolog programs. On the other hand, they can change the declarative meaning of a program. Following [80], a cut always succeed and commits Prolog to all the choices made since the parent goal was unified with the head of the clause the cut occurs in. As a consequence, a cut prunes all the clauses that are below it and all alternatives solutions to the conjunction of goals appearing at its left.

Let us briefly explain what cuts are. As we mentioned earlier, the mechanism that Prolog uses for finding multiple solutions is *backtracking*.

Consider for instance the following program

```
1. p(X)   :-a(X).
2. p(X)   :-b(X).
3. p(X)   :-c(X).
4. a(1).
5. b(2).
6. c(3).
```

The goal p(X) gives the solutions X=1, X=2, and X=3. The execution tree is shown in Figure 4.6.

Now let us introduce a cut in the second clause of the program, namely:

```
1. p(X)   :-a(X).
2. p(X)   :-b(X), !.
3. p(X)   :-c(X).
4. a(1).
5. b(2).
6. c(3).
```

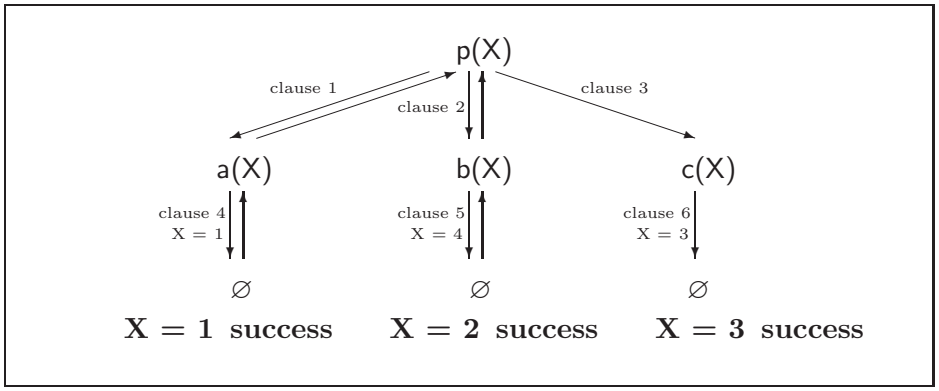


Fig. 4.6: The execution tree of a cut-free program.

The goal p(X) succeeds again, but now with solutions X=1 and X=2. The branch leading to the third solution has been pruned away as shown in Figure 4.7.

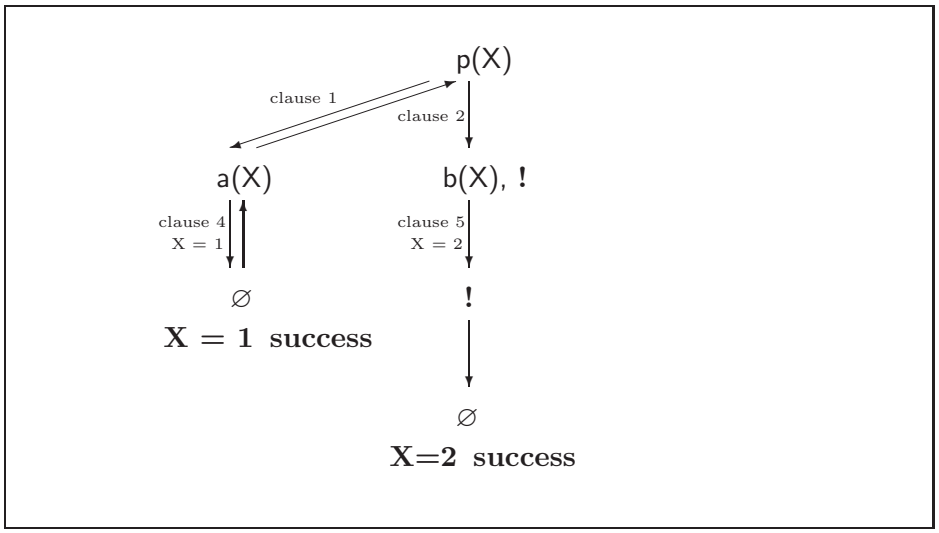


Fig. 4.7: The execution tree of a program with a cut.

Here the clauses below the cut have been pruned away. The unification X = 3 is ruled out by the cut and the only solutions that have been considered are X = 1 and X = 2.

To see what happens with alternative solutions at the left of the cut, we consider a program in two versions, without and with cut.

Next we see the effect of cuts on the set of alternative solutions.

Version without cut	**Version with cut**
1. p(X) :- a(X).	1. p(X) :- a(X), !.
2. a(1).	2. a(1).
3. a(2).	3. a(2).

The respective execution trees for the goal p(X) are shown in Figure 4.8.

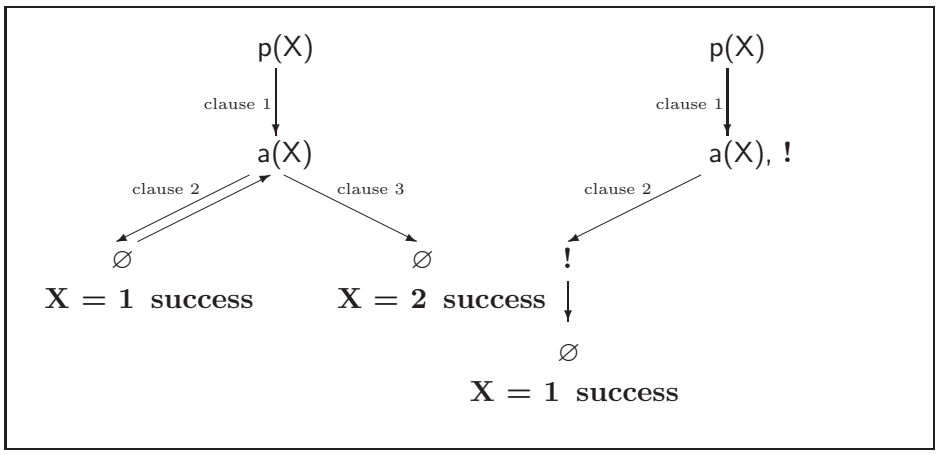


Fig. 4.8: The execution trees for a simple program without and with cut.

The cut has eliminated all alternative solutions to the conjunction of goals at its left. In the examples we have seen the cuts change the set of solutions. This is not necessarily the case. Consider the following program for merging two ordered lists, taken from [80]:

```
merge([H1|T1],[H2|T2],[H1|T3])  :-  >(H1,H2), !,
                                    merge(T1,[H2|T2],T3).
merge([H1|T1],[H2|T2],[H2|T3])  :-  =<(H1,H2), !,
                                    merge([H1|T1],T2,T3).
merge(L1,[ ],L1)                :-  !.
merge([ ],L2,L2)                :-  !.
```

It is clear that exactly one of the conditions, either H1 $>$ H2 of H1 $\leq$ H2 holds. In this case, the cuts are used to prevent Prolog from searching useless paths. Since only one of the clauses of the merge-program may be chosen at a given point, it makes no sense to continue analysing the others, and the cuts prune all the clauses under them. This is a case in which the use of cuts does not change the set of solutions. These cuts are sometimes called *green cuts* [80]. Cuts that change the set of solutions are called *red cuts*. In our implementation, green and some red cuts have been used. We will mention the most relevant ones.

4.3 Usage of the Program

The program is divided in several modules, namely main.pl, mod_reports.pl, mod_services.pl, mod_parse.pl, and mod_proof.pl. All these files are text files that may be opened and modified with any text editor. The file main.pl has the possibility of setting the number of agents by choosing which of the agents/1 clauses is active (exactly one must be active; all others must be set as comments. This is done by preceding the line with the symbol '%') For instance, the following is the setting for one single agent:

```
%agents(['1','2','3']).
%agents(['1','2']).
agents(['1']).
```

Suppose for the sake of the example, that these files are in the directory C:\Documents and Settings\UserName\My Documents\Prolog\CK\v.10. To run the program, we start SWI-Prolog and write the goal

```
1 ?- ['C:/Documents and Settings/UserName/My Documents/Prolog/
CK/v.10/main.pl'].
```

This goal loads the program and all its modules. It does not compile the program (Prolog is an interpreted language.) If some change is introduced in the program during the session (for instance the number of agents is set to two instead of one), the program must be reloaded.

After we have entered the goal above, we get:

```
%  mod_parse compiled into parse 0.00 sec 10,512 bytes.
%  mod_proof compiled into proof 0.00 sec 14,652 bytes.
%  mod_services compiled into services 0.00 sec 6,024 bytes.
%  mod_reports compiled into reports 0.00 sec 23,420 bytes.
% C:/Documents and Settings/UserName/My Documents/Prolog/
CK/v.10/main.pl compiled 0.00 sec 86,588 bytes
true.
```

The execution of the program is triggered with the predicate prove/2, whose first argument is the sequent to be proved or disproved and the second one is the name of a text file where the session's log will be stored.

For instance, suppose we try the prove the sequent $(p \vee q), (\neg q \wedge \neg p)$ and to put the log of the proof in the file P01.txt. Assuming that the program is set for one single agent, we write the goal

```
2 ? - prove('(p V q),(¬q & ¬p)','P01').
true.
```

The sequent is clearly valid. The program delivers a proof of it, that is stored in the file P01.txt, located in the same directory as the programs. The contents of this file has three components: the syntactic parsing, the preproof, and the conclusion. The first part is the following:

The next part is the preproof. Each node is assigned a number (which corresponds to the code of it in the tree) and the successors are indicated by the respectives codes.

The preproof is shown as depicted in Figure 4.10:

Now let us try a sequent that is not valid, namely $\Diamond\Box_1\neg p, \Box_1\neg p$. We want the program to write the log in file P02.txt. We write the goal

```
Agents     [1]
Sequent    [(p V q), (¬q & ¬p)]

Formula Nr. 1: (p V q)
OK

Formula Nr. 2: (¬q & ¬p)
OK

** Parsing OK. Proof-search goes on.
```

Fig. 4.9: The parsing part of a simple proof.

```
Formula Nr. 1: (p V q)
OK

Formula Nr. 2: (¬q & ¬p)
OK

** Parsing OK. Proof-search goes on.

Node <0> = [(p V q), (¬q & ¬p)]
OR rule yields node <1>

Node <1> = [p, q, (¬q & ¬p)]
AND rule yields nodes <2> and <3>

Node <2> = [¬q, p, q]
instance of ID -- Node successful

Node <3> = [¬p, p, q]
instance of ID -- Node successful

** Total number of nodes          = 4
** Maximum height of the tree     = 3
** Time elapsed                   = 0e-06 sec

** The sequent is valid
```

Fig. 4.10: A simple proof.

```
4 ?- prove('UK1¬p, K'¬p','P02').
true.
```

In this case, the program provides a countermodel to the given sequent. The log contains three parts. The first one is the parsing part of the file

P02.txt, which is very similar to the previous one and is omitted here. It may be seen in Chapter 5The next part corresponds to the preproof. As in the previous example, the tree is developed until no further progress is possible.

The last part of the log contains a countermodel derived from the failed preproof. The countermodel is constructed following the procedure sketched in the completeness proof of Chapter 3 (see Definition 3.49.) This is the most complicated part of the program; the proof procedure is relatively simple and straightforward in comparison. We show next the preproof.

Observe that the cyclic nodes <7> and <11> have been detected and singled out. After the detection of a cyclic node the branch is not further explored. The countermodel part is shown next.

Observe that the countermodel (S, R, v) obtained has three states $S = \{s_0, s_1, s_2\}$, and the relations for agent 1 are (s_0, s_1), (s_1, s_2), and (s_2, s_2). The relation set for agent 2 is empty. Finally, the valuations are as follows: $v(s_0) = \varnothing$, $v(s_1) = v(s_2) = \{p\}$.

4.4 Representation of Formulæ and Sequents

Now we begin with the description of the actual implementation of S'_{CK}. The program is a prototype and as such it has a plain user interface which uses ASCII characters. Therefore we do not have at our disposal subindices and symbols like $\Box$, $\Diamond$, $\Box\!\!\!\!\Diamond$ or $\boxtimes$. Several choices were possible. We begin by describing the actual implementation and comment on further possibilities at the end of the section.

There are two main representations of formulæ and sequents. The normal form 0, abbreviated by NF0 is the representation used in the user interface. The normal form 2 is the internal representation used by the program. The normal form 1 is only an intermediate representation used when converting from NF0 to NF2. This is illustrated in Figure 4.13.

The normal form 0 is a string of characters and is used to represent sequents in a (more or less) readable form. For instance, the sequent

Node <0> = [UK1¬p, K2¬p]
U rule yields node <1>

Node <1> = [K1¬p, (P1UK1¬p V P2UK1¬p), (P1K1¬p V P2K1¬p)]
OR rule yields node <2>

Node <2> = [K1¬p, (P1K1¬p V P2K1¬p), P1UK1¬p, P2UK1¬p]
OR rule yields node <3>

Node <3> = [K1¬p, P1UK1¬p, P1K1¬p, P2UK1¬p, P2K1¬p]
BOX rule yields node <4>

Node <4> = [¬p, UK1¬p, K1¬p]
U rule yields node <5>

Node <5> = [¬p, K1¬p, (P1UK1¬p V P2UK1¬p), (P1K1¬p V P2K1¬p)]
OR rule yields node <6>

Node <6> = [¬p, K1¬p, (P1K1¬p V P2K1¬p), P1UK1¬p, P2UK1¬p]
OR rule yields node <7>

Node <7> =[¬p, K1¬p, P1UK1¬p, P1K1¬p, P2UK1¬p, P2K1¬p]
BOX rule yields node <8>

Node <8> =[¬p, UK1¬p, K1¬p]
U rule yields node <9>
Node <9> =[¬p, K1¬p, (P1UK1¬p V P2UK1¬p), (P1K1¬p V P2K1¬p)]
OR rule yields node <10>

Node <10> =[¬p, K1¬p, (P1K1¬p V P2K1¬p), P1UK1¬p, P2UK1¬p]
OR rule yields node <11>

Node <11> = [¬p, K1¬p, P1UK1¬p, P1K1¬p, P2UK1¬p, P2K1¬p]
cyclic node: twin node is <7> -- Node unsuccessful

Fig. 4.11: The tree of the second example.

$\{\boxed{*}p, (\neg p \wedge (q \vee r)), \Diamond_1 \diamondsuit q\}$ is represented as

['Cp, (p & (q V r)), P1Uq']

The normal form 1 is an intermediate form. It is just a list of lists, one for each formula of the sequent. The internal lists are sequences of the characters of the corresponding NF0 representation without spaces. The same sequent is represented as follows in NF1:

```
** Countermodel = (S, R, v)

S = {s0, s1, s2}

R = {R1, R2}
  R1 = {(s0, s1), (s1, s2), (s2, s2)}
  R2 = {}

v = {(s0, []),
   (s1, [p]),
   (s2, [p])}
** Total number of nodes          = 12
** Maximum height of the tree     = 12
** Time elapsed                   = 0e-06 sec

** The sequent is not valid.
```

Fig. 4.12: The countermodel of the second example.

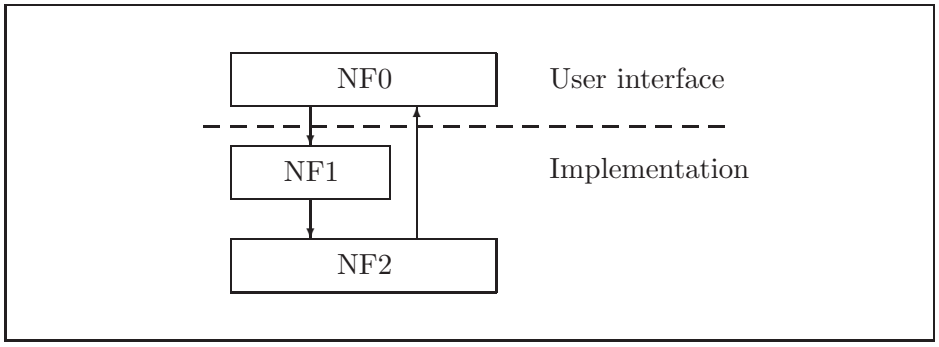


Fig. 4.13: The different representations of formulæ and sequents.

```
[['C', p], ['(', p, &, '(', q, 'V', r, ')', ')'], ['P', 1, 'U', q]]
```

Single quotation marks indicate that we are working with strings and not variables. In order to process the formulæ, a Prolog term that reflects the internal structure of it must be constructed. NF2 is the form in which sequents are internally processed. The sequent we have considered above is represented in NF2 as

[c(plit(p)), y(plit(p), o(plit(q), plit(r))), p(1, u(plit(q)))]

A sequent is thus expressed as a list of terms where each formula corresponds to a term. The correspondence between the different representations is shown in Figure 4.14.

Formula	NF0 notation	NF2 notation
p	'p'	plit(p)
$\neg p$	'¬p'	nlit(p)
$(\varphi \vee \psi)$	'('φ_{NF0}' V 'ψ_{NF0}')'	o(φ_{NF2},ψ_{NF2})
$(\varphi \wedge \psi)$	'('φ_{NF0}' & ' ψ_{NF0}')'	y(φ_{NF2},ψ_{NF2})
$\Box_i\varphi$	'K'$i\varphi_{\mathsf{NF0}}$	k(i,φ_{NF2})
$\Diamond_i\varphi$	'P'$i\varphi_{\mathsf{NF0}}$	p(i,φ_{NF2})
$\boxtimes\varphi$	'C'φ_{NF0}	c(φ_{NF2})
$\circledast\varphi$	'U'φ_{NF0}	u(φ_{NF2})

Fig. 4.14: The representation of formulæ in NF0 and NF2.

Notation: we denote by φ_{NF0} and φ_{NF2} the representations of a formula φ in NF0 and NF2 respectively. If T1 and T2 are strings, we abbreviate by $\mathsf{T1}\varphi_{\mathrm{NF0}}\mathsf{T2}$ the string resulting from the concatenation of the strings T1, φ_{NF0}, and T2.

It is not hard to see that the NF2 notation is not very readable, even when simple formulæ and sequents are represented. As commendet above, the data are entered by the user in NF0 format and the program shows them also in this form. The internal conversion and parsing procedures are explained in the next section. We will omit the single quotation marks whenever no confusion arises.

We comment now briefly on other alternatives. The main option was to define the operators as infix and to use one single representation throughout the process. There are some advantages with this approach. The main one is that no conversion between the different forms are needed and that the

parsing of formulæ becomes also unnecessary. Prolog returns an error code if the term is not a correct one (recall that a formula in NF0 is a string and thus it is always a correct term, even when the represented formula is not well-formed.)

Nevertheless there are also some drawbacks when we use a single representation. The main one is the difficulty to use uppercase letters (V, K, P, C, and U) as functors. Prolog reserves all strings beginning with uppercase letters as variables. We are thus faced with two possibilities:

1. We quote the uppercase letters. In this case, it is clear that we are not referring to a variable. The sequent we have been considering would then look thus:

$$['C'\ p\ ,(p\ \&\ (q\ 'V'\ r)),'P'(1,'U'\ q)]$$

2. We may use lowercase letters. The problem is the possible confusion between letters representing connectors and letters representing atomic propositions. The sequent has in this case the following aspect:

$$[c\ p,(p\ \&\ (q\ v\ r)),p(1,u(q))]$$

The latter option is probably best, although we did not find it entirely satisfying.

To ease the reading of this chapter, where the implementation of some predicates is briefly explained, we adopt the following convention: predicates will be written in NFo, although they are in NF2 in the actual implementation, and we treat them as if they were infix operators. For instance, the clause:

```
subformula(o(X,Y),[o(X,Y)|T])  :-  subformula(X,T1),
                                    subformula(Y,T2),
                                    append(T1,T2,T)
```

will be written as follows:

```
subformula((X V Y),[(X V Y)|T])  :-  subformula(X,T1),
                                     subformula(Y,T2),
                                     append(T1,T2,T)
```

The way in which infix operators may be defined is explained in Appendix C.

4.5 Parsing and Converting Formulæ

The first task is the parsing of the sequent entered by the user. This process is performed simultaneously with the conversion of this sequent into NF2. The parsing clauses control that the formula agrees with the syntax of Definition 3.8. It is a rather strict implementation: not even redundant parentheses are tolerated, and all parentheses prescribed by the grammar must be included. In other words, formulæ like

$$\varphi \wedge \psi \vee \xi \qquad (\mathsf{C}(\alpha \vee \beta))$$

are not accepted: they must be respectively written as

$$((\varphi \wedge \psi) \vee \xi) \qquad \mathsf{C}(\alpha \vee \beta)$$

The main part of the parsing module is the set of clauses defining the predicate parse_formula/3, where the first parameter is the formula in NF1, the second one is the formula in NF2, and the third one is an error code (0 means that the parsing was successful and 1 means that syntax errors were found.) The predicate is transcribed in Appendix A. It reflects the tree-like nature of the term representing a formula. The parser determines the tree-structure of the formula in NF1 and produces the corresponding formula in NF2 as shown in the example of Figure 4.15.

Observe that the only difference in the shape of the trees is the edge below the negation. This is because the atomic expresions in NF2 are literals and not plain atomic propositions.

The determination of the main connector is easy in the case of connectors $\boxplus$, $\diamondplus$, $\Box_i$ and $\Diamond_i$, since it it s the head of the list. In the case of $\wedge$ or

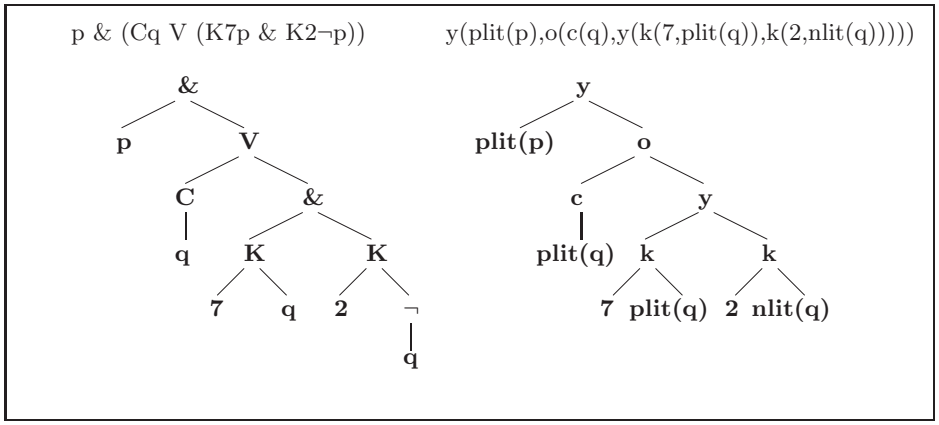


Fig. 4.15: The tree structures of a formula in NF0 and NF2.

$\vee$, it is a little bit more complicated. To find the main connector we use the *parentheses depth*, defined next.

Definition 4.4 (Parentheses-depth) *Let L be a list and let X be an element of the list. The* parentheses-depth of X in L*, or* p-depth of X in L *for short, is defined as the number of left parentheses minus the number of right parentheses in L at the left of X.*

The main connector in a formula beginning with a left parenthesis is the one with the least p-depth. For example, assume we have a formula like

$$(((\mathsf{p} \vee \mathsf{q}) \wedge (\mathsf{r} \wedge \neg\mathsf{s})) \vee ((\neg\mathsf{p} \wedge \mathsf{q}) \wedge \mathsf{p}))$$

The NF1 expression of this formula is the following list. The p-depths of the connectors are indicated above them.

$$['(','(','(',\mathsf{p},\overset{3}{\text{'V'}},\mathsf{q},')',\overset{2}{\text{'\&'}},'(',\mathsf{r},\overset{3}{\text{'\&'}},\neg,\mathsf{s},),),\overset{1}{\text{'V'}},'(','(','(',\neg,\mathsf{p},\overset{3}{\text{'\&'}},\mathsf{q},')',\overset{2}{\text{'\&'}},\mathsf{p},')',')']$$

Observe that if a list represents a formula in NF1, no element of it may have negative p-depth. Besides, in any formula beginning with a left parenthesis there is exactly one connector $\vee$ or $\wedge$ having p-depth 1.

When a formula has to be shown to the user, it is reconverted from NF2 to NF0. This is done by the predicate term2string/2, where the first argument is a NF2 expression and the second one the corresponding NF0 expression. This conversion is simpler than the other one. The complete listing is in Appendix A.

4.6 Construction of a Proof in S_{CK}

As shown in the first example of Section 4.5, a successful proof yields a proof-tree in S_{CK}. During the process of proof-search (see Sections 4.7 and 4.8 for a detailed description of it), the dynamic table deriv/9 is constructed. This table contains a complete description of the derivation tree. Each entry of this table describes a node, successful or not, of the proof. The general form of this clause is deriv(V,Seq,Hst,K,Rule,HF,R,Succ,H) where:

- V is a list of positive literals plit(X) such that the node contains nlit(p). This list is used only in the construction of a countermodel (see Section 4.6) and corresponds to the valuation v of it.

- Seq and Hst are representations in NF0 of the sequent and its annotation, if any.

- K is the code of the node in the S'_{CK} proof.

- Rule is the rule that was backwards-applied to the conclusion. If the node is irreducible, it has the value i; if it is cyclic, it has the value o.

- HF is a variable that indicates whether the sequent is history-free (0) or annotated (1).

- R indicates whether the node is successful (0) or not (1).

- Succ has either the codes of nodes with the premises in the case of rules other than $\Box'$, or a list with terms of the form ap(A,Kx) where A is an agent and Kx is the code of the i- premise node in the case of rule $\Box'$.

- H is the height of the node measured in number of nodes.

The construction of the proof proceeds by scanning the tree from the root upwards. All premises of rules other than $\Box'$ and one successful premise of rule $\Box'$ are chosen. This is implemented in module mod_reports. This module writes a text file with the details of the derivation, and it is called with argument (OO) where OO is the output stream.

Recall that the proof is actually a proof in S'_{CK}. The construction of the proof in S_{CK} starting from the proof in S'_{CK} is realised by the predicate get_derivation/1. Starting from the root of the derivation in *ssck* (the node with code 0, the child nodes are obtained directly with the variable Succ in the case of rules other than $\Box'$ and, in the latter case, the first successful branch starting fromt the left is chosen. A branch is closed when an axiomatic node has been found. The process goes on until all branches are closed. The main difference between proofs in S_{CK} and proofs in S'_{CK} is the branching out of rule $\Box'$ in the latter. This is an "or-rule" in the sense that it is enough that a single premise be satisfied for the conclusion to be satisfied. In the case an instance of $\Box'$ is encountered, a corresponding instance of $\Box$ is derived by choosing the leftmost successful premise.

There is also the construction of the preproof when the proof is not successful. The process is very similar to this one, with some slight differences. The complete code is in Appendix A.

4.7 Construction of a Countermodel in S_{CK}

In the case that the sequent is not provable, a countermodel is constructed following Definition 3.49 of Chapter 3. We have already seen an example with a sequent which is not valid in Section 4.3.

In this section we succinctly explain the construction of the countermodel. The technical part is rather involved and a detailed explanation is in Appendix B.

We first informally recall the procedure of construction of a model by means of a simple example. The actual definition is in Chapter 3 (see Definition 3.49.) We are interested only in the structure of the failed preproof. With this in mind we represent the sequents as nodes in a tree and the edges are the application of the rules. Instances of all rules but $\Box'$ are represented by simple lines. Instances of $\Box'$ rules are represented by lines cut by a short segment and a specification of the agent i in the case of an i-premise. The edges corresponding to instances of the rule $\Box'$ have been singled out as shown in Figure 4.16

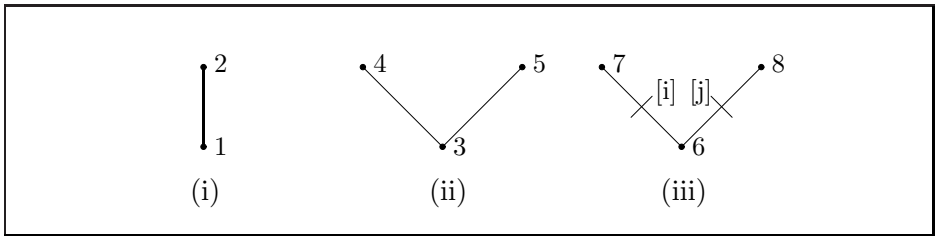


Fig. 4.16: The representation of the nodes of a preproof.

Here, (i) is a rule with one single premise, (ii) represents a rule with two premises and (iii) represents an instance of $\Box'$, where 7 is an i-premise of 6 and 8 is a j-premise of 6. We give now an example of the algorithm to obtain a countermodel from a failed preproof. Consider the preproof of Figure 4.17.

First, a tree τ_1 is constructed by going upwards in the preproof and keeping the leftmost unsuccessful branch in any instance of a branching rule other than $\Box'$. See Figure 4.18, where discarded branches are in grey.

The second part is the construction of the tree τ_2 by collapsing all nodes that are connected and not separated by an instance of $\Box'$.

With this tree, shown in the left part of Figure 4.19, the model can be constructed. The states and the relations are shown in the right part of the

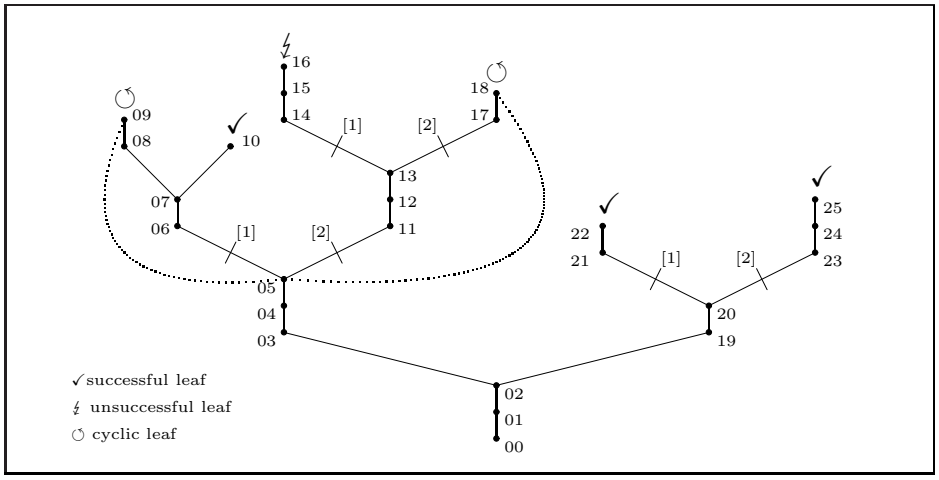


Fig. 4.17: The structure of a failed preproof.

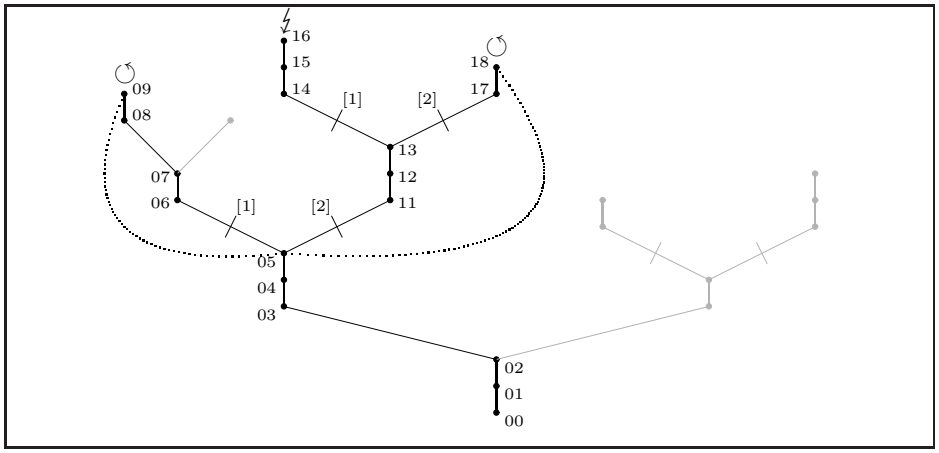


Fig. 4.18: The first step in the construction of a countermodel.

same figure.

Observe that $(s_1, s_1), (s_4, s_1) \in R_1$ and $(s_1, s_2), (s_4, s_2) \in R_2$. This is because the nodes 05 and 09 are twin nodes, as are the nodes 05 and 18.

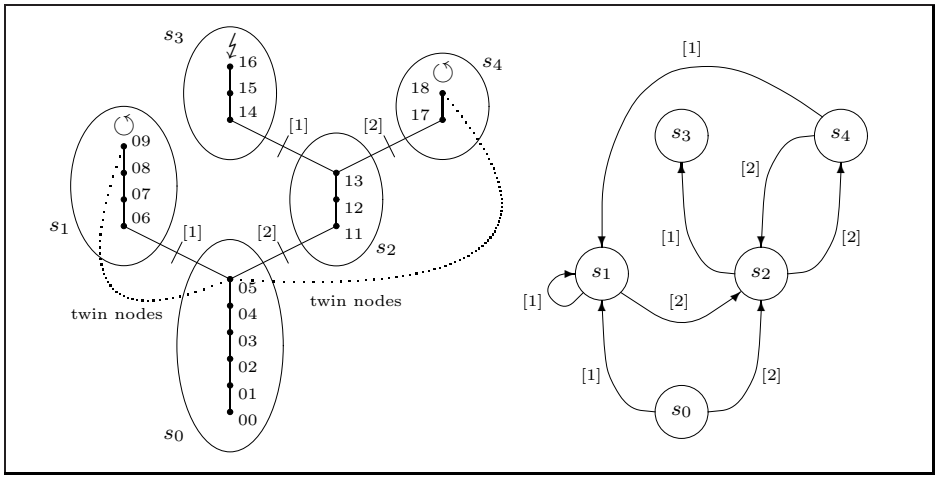


Fig. 4.19: The last part of the construction of the countermodel (left) and the resulting model (right.)

The node 06 is a 1-premise of 05 and the node 11 is a 2-premise of 05. These nodes are collapsed according to the following table:

Nodes	State
00–05	s_0
06–09	s_1
11–13	s_2
14–16	s_3
17–18	s_4

Thus, according to Definition 3.49, there must be arrows labelled [1] from the states containing 08 (s_1) and 18 (s_4) to the states containing the 1-premises of 05 and arrows labelled [2] from the former states to the states containing the 2-premises of 05. The 1-premise of node 05 is node 06, contained in state s_1 and its 2-premise is node 11, contained in state s_2. Hence, there are arrows labelled [1] from s_1 to itself and from s_4 to s_1 and arrows labelled [2] from s_1 and s_4 to s_2 (see right part of Figure 4.19.) Again, the basis is the table deriv/8, which stores the whole information

of the preproof. The extraction of the derivation is in module mod_reports as is the construction of a proof in Section 4.5. The construction of a countermodel is a more complicated process than the construction of a proof. We will give a general explanation without many technical details.

The first part is the construction of a premodel, which consists of two lists:

- A list St of terms of the form st(N,P,Kx), where N is the code of the state in the countermodel, P is a list of the propositions that hold in the state according to Definition 3.49, and Kx is a list containing codes of the nodes of the preproof that are collapsed in the state N.
- A list Re of terms br(A,K1,K2) where A is an agent, and K1 and K2 are codes of nodes in the preproof such that K2 is an A-premise of K1.

The premodel contains the whole information of the model. Having the lists St and Re, the extraction of the model is routine. The actual model consists of:

- A list Ms of states s0, s1, and so on.
- A list Mv of valuations, containing terms of the form v(S,P) where S is a state of the list Ms and P is a list of atomic propositions.
- A list Mr of lists of terms r(S1,S2), where S1 and S2 are states of the list Ms. The i-th list is the list of relations of agent i.

This is done with the predicates get_sv/3 and get_r/4. The whole listing is in Appendix A and a detailed explanation of the process of construction of a countermodel is in Appendix B.

4.8 Implementation of the Rules of S'_{CK}

In this section we show how the rules and the proof-search process of system S'_{CK} are implemented. The implementation of the decision procedure is in

module mod_proof and may be consulted in Appendix A. In this section we will use infix notation as explained above and we will assume that the operators are implemented with the NF0 syntax. This is done to ease the reading adnd does not correspond to the actual implementation as shown in the aforementioned appendix.

We begin with the axiom id':

$$\mathsf{id}'\ \frac{}{\Gamma,\ p,\ \neg p\ :\ l}$$

All we have to do is to determine whether there are two complementary literals in the sequent.

Recall that the sequents are expressed by a list L of formulæ in NF2. To implement this rule we have to test whether there are two elements plit(X) and nlit(X) for some X in L. In NF0, these literals are denoted by P and ¬P. The implementation is in Figure 4.20.

```
id_rule(L) :- member(P,L), member(¬P,L).
```

Fig. 4.20: The implementation of rule id′.

The simplest rule with one single premise is $\vee'$:

$$\vee'\ \frac{\Gamma,\ \varphi,\ \psi\ :\ l}{\Gamma,\ \varphi \vee \psi\ :\ l}$$

This rule is implemented as shown in Figure 4.21.

The predicate no_dup(L1,L2) succeeds when L2 contains the same elements as L1 but with no duplicates. It is in module mod_services. The sort_list/2 predicate is use here and in the following rules to have "normalised" sequents so as to simplify their comparison.

The predicate o_rule1(L1,L2) is recursively called (clause 4) until a disjunction appears at the head of L1 (clause 3.) Then both disjuncts are

```
1.  o_rule(L1,L2)                    :-  o_rule1(L1,L2a), no_dup(L1a,L2b),
                                         sort_list(L2b,L2).

2.  o_rule1([],_)                    :-  !, fail.
3.  o_rule1([(X V Y)|T],[X,Y|T])     :-  !.
4.  o_rule1([H|T1],[H|T2])           :-  !, o_rule1(T1,T2).
```

Fig. 4.21: The implementation of rule $\vee'$.

extracted and put in L2. If there is no disjunction in L1, clause 2 is reached and the predicate fails. The cut in clause 2 is a green cut. It does not change the set of solutions, since clause 2 is only reached when no disjunction (i.e., no term o(X,Y)) is in the first argument. It is there to avoid useless backtracking.

The simplest rule with two premises is $\wedge'$.

$$\wedge' \ \frac{\Gamma,\, \varphi \ : \ l \qquad \Gamma, \psi \ : \ l}{\Gamma,\, \varphi \wedge \psi \ : \ l}$$

The implementation of this rule is in Figure 4.22.

```
1.  y_rule(L1,L2,L3)                 :-  y_rule1(L1,L2a,L3a),
        no_dup(L2a,L2b), no_dup(L3a,L3b),
        sort_list(L2b,L2), sort_list(L3b,L3).

2.  y_rule1([],_,_)                  :-  !, fail.
3.  y_rule1([(X & Y)|T],[X|T],[Y|T]) :-  !.
4.  y_rule1([H|T1],[H|T2],[H|T3])    :-  !, y_rule1(T1,T2,T3).
```

Fig. 4.22: The implementation of rule $\wedge'$.

The process is very similar to the previous one, but with two premises

instead of one. The predicate y_rule(L1,L2,L3) is recursively called (clause 4) until a conjunction appears at the head of L1 (clause 3.) Then the two conjuncts are extracted and respectively put in L2 and L3. If there is no conjunction in L1, clause 2 is reached and the predicate fails.

The other rules are more complex. In the case of the rules $\circledast'$ and $\boxtimes'$, the respective occurrences of $\Diamond$ and $\Box$ in the premises make them dependent on the number of agents: for instance, if there is one single agent ($\mathcal{A} = \{1\}$) the premises of $\boxtimes\varphi$ are $\Box_1\varphi$ and $\Box_1\boxtimes\varphi$; if there are two agents ($\mathcal{A} = \{1, 2\}$), the premises of the same formula are $(\Box_1\varphi \wedge \Box_2\varphi)$ and $(\Box_1\boxtimes\varphi \wedge \Box_2\boxtimes\varphi)$. The same happens with a formula of the form $\circledast\varphi$.

Recall the rule $\circledast'$:

$$\circledast' \frac{\Gamma,\ \Diamond\circledast\varphi,\ \Diamond\varphi\ :\ l}{\Gamma,\ \circledast\varphi\ :\ l}$$

This rule is implemented as shown in Figure 4.23.

```
1. u_rule(L1,L2)                          :- u_rule1(L1,L2a), no_dup(L2a,L2b),
                                             sort_list(L2b,L2).
2. u_rule1([],_)                          :- !, fail.
3. u_rule1([U(X)|T],[E1,E2|T])            :- !, u_unfold(X,E1,E2).
4. u_rule1([H|T1],[H|T2])                 :- !, u_rule1(T1,T2).
5. u_unfold(X,E1,E2)                                    :-
                                             agents(L),_unfold(X,E1,E2,L).
6. u_unfold(X,P(N,X),P(N,U(X)),[N])                     :- !.
7. u_unfold(X,(P(N,X) V Z1),(P(N,U(X)) V Z2),[N|T])     :- !,
                                             u_unfold(X,Z1,Z2,T).
```

Fig. 4.23: The implementation of rule $\circledast'$.

The change with respect to a predicate like o_rule is the introduction of the u_unfold/3 and u_unfold/4 predicates. The idea of the call to predicate u_unfold(X,E1,E2) is the following: if $\circledast\varphi$ is the active formula in the rule,

then X is instantiated to φ, and after the call to u_unfold/3, we have the following instantiations:

$$\begin{array}{lcl} \mathsf{E1} & = & ((\ldots(\Diamond_1\varphi \vee \Diamond_2\varphi) \vee \ldots) \vee \Diamond_n\varphi) \\ \mathsf{E2} & = & ((\ldots(\Diamond_1\!\!\Diamond\varphi \vee \Diamond_2\!\!\Diamond\varphi) \vee \ldots) \vee \Diamond_n\!\!\Diamond\varphi) \end{array}$$

In both cases for all agents $1, 2, \ldots, n \in \mathcal{A}$. This is shown next.

Fact 4.5 *Let* X *be the NF2 representation of a formula* φ *and let* L = [1, 2, ... k] *be a list of agents. Then* u_unfold(X,E1,E2,L) *succeeds if* E1 *and* E2 *are NF2 representation of the disjunctions*

$$((\ldots(\Diamond_1\varphi \vee \Diamond_2\varphi) \vee \ldots) \vee \Diamond_k\varphi),$$

$$((\ldots(\Diamond_1\!\!\Diamond\varphi \vee \Diamond_2\!\!\Diamond\varphi) \vee \ldots) \vee \Diamond_k\!\!\Diamond\varphi).$$

Proof. Induction on the length of L.

Base case: the base case (L=[1], the signature has one single agent) is immediate: clause 6 succeeds and we get u_unfold(X,p(1,X),p(1,u(X)),[1]).

Induction step: if L=[N|T], then by induction hypothesis we have that u_unfold(X,E1',E2',T) if E1' and E2' are NF2 representations of the disjunctions $\Diamond_1\varphi \vee \ldots \vee \Diamond_{k-1}\varphi$ and $\Diamond_1\!\!\Diamond\varphi \vee \ldots \vee \Diamond_{k-1}\!\!\Diamond\varphi$. This implies that clause 7 succeeds and we get u_unfold(X,o(p(1,X),E1'), o(p(1,u(X)),E2'),L), where the second and third arguments are NF2 representations of the disjunctions $\Diamond_1\varphi \vee \ldots \vee \Diamond_k\varphi$ and $\Diamond_1\!\!\Diamond\varphi \vee \ldots \vee \Diamond_k\!\!\Diamond\varphi$. ■

The way the predicates u_rule/2 and u_rule1/2 work is quite similar to the other predicates we saw before. The predicate u_rule1/2 is recursively called (clause 4) until a formula of the form u(X) appears (clause 3.) Then the two disjunctions for all agents are extracted with predicate u_unfold/3 and put in the premise. If there is no formula of the form u(X) in L1, clause 2 is eventually reached and the predicate fails.

Let us consider now the rule $\boxdot'$:

$$\boxtimes' \frac{\Gamma,\, \Box\varphi \,:\, l, \quad \Gamma,\, \Box\boxtimes\varphi \,:\, l}{\Gamma,\, \boxtimes\varphi \,:\, l}$$

As it was the case with the predicate $\diamondsuit'$, we have to "unfold" the formula $\boxtimes'\varphi$ so as to get instances of $\Box_i'\varphi$ for all agents i. This is done with the predicates c_unfold/3 and c_unfold/4, which are duals of the predicates u_unfold/3 and u_unfold/4 described above.

If $\boxtimes\varphi$ is the active formula in the rule, then X is instantiated to φ, and a call to c_unfold(X,E1,E2) yields the following instantiations:

$$\begin{aligned} \mathsf{E1} &= ((\ldots(\Box_1\varphi \wedge \Box_2\varphi) \wedge \ldots) \wedge \Box_k\varphi) \\ \mathsf{E2} &= ((\ldots(\Box_1\boxtimes\varphi \wedge \Box_2\boxtimes\varphi) \wedge \ldots) \wedge \Box_k\boxtimes\varphi) \end{aligned}$$

In both cases for all agents $1, 2, \ldots, k \in \mathcal{A}$.

The rule $\boxtimes'$ is implemented as shown in Figure 4.24.

```
1. c_rule(L1,L2)                    :- c_rule1(L1,L2a), no_dup(L2a,L2b),
                                       sort_list(L2b,L2).

2. c_rule1([],_)                    :- !, fail.
3. c_rule1([c(X)|T],[E1,E2|T])      :- !, c_unfold(X,E1,E2).
4. c_rule1([H|T1],[H|T2])           :- !, c_rule1(T1,T2).

5. c_unfold(X,E1,E2)                                  :- agents(L),
                                                         c_unfold(X,E1,E2,L).
6. c_unfold(X,k(N,X),k(N,u(X)),[N])                   :- !
7. c_unfold(X,y(k(N,X),Z1),y(k(N,u(X)),Z2),[N|T])     :- !, c_unfold(X,Z1,Z2,T).
```

Fig. 4.24: The implementation of rule $\boxtimes'$.

The correction of the predicate c_unfold/3 is shown next.

Fact 4.6 *Let* X *be the NF2 representation of a formula* φ *and let* L = [1, 2, ... *k*] *be a list of agents. Then* c_unfold(X,E1,E2,L) *if* E1 *and* E2 *are NF2*

representation of the conjunctions

$$((\ldots(\Box_1\varphi \wedge \Box_2\varphi) \wedge \ldots) \wedge \Box_k\varphi),$$

$$((\ldots(\Box_1\circledast\varphi \wedge \Box_2\circledast\varphi) \wedge \ldots) \wedge \Box_k\circledast\varphi).$$

Proof. Dual to the proof of Fact 4.5. ■

The $\Box'$ rule has the extra complication that we do not know *apriori* how many premises a conclusion may have. The rule is shown below:

$$\Box' \ \frac{\Sigma_1 : l \quad \ldots \quad \Sigma_q : l}{\Gamma : l} \quad \text{where} \begin{cases} \Gamma \text{ is locally reduced, and} \\ \Sigma_i \in \{\Sigma_1, \ldots, \Sigma_q\} \text{ iff} \\ \quad \frac{\Sigma_i}{\Gamma} \text{ is a good instance of } \Box_i \end{cases}$$

In other words, if we start with a conclusion Γ, for each formula of the form $\Box_i\psi$ in φ there is a premise ψ, Δ, where Δ consists of all formulæ ζ such that there is a formula of the form $\Diamond_i\zeta$ in Γ. In the implementation `L` represents Γ and `k(A,X)` represents $\Box_i\psi$.

The rule $\Box'$ is processed in three parts. First there is a test to verify that the sequent `L` is a NF2 representation of a locally reduced sequent and contains at least one formula of the form `k(A,X)`. This is done by the predicate `is_locally_reduced/1`. Second, the collections of all formulæ of the form `k(A,X)` or `p(A,X)` are gathered. This is done by the predicate `get_kp/3`. The third and final part is the construction of the list of all good premises (see Definition 3.15.) It is performed with the predicate `get_premises/5`.

The main predicate is `k_rule(L,Lx,N)` where `L` is the conclusion. The term `Lx` is to be instantiated with a list of terms of the form `ap(A,Ls)`, where `A` is an agent and `Ls` is a premise of `L` by rule $\Box'$, and `N` is to be instantiated with the total number of premises. We do not use just a list of premises as in the other rules because we need the information of the i-successors (see Section 3.6) to be able to eventually construct a countermodel. Incidentally, in tableaux-systems this information is also necessary but to track "unfulfilled eventualities" [1, 78].

```
1.  k_rule(L1,Lx,N)   :-  locally_reduced(L), !, get_kp(L,L1,L2),
                          get_premises(L1,L2,Lxa,0,N), sort_list2(Lxa,Lx).
2.  sort_list2([],[]).
3.  sort_list2([ap(A,L1)|T1],[ap(A,L2)|T2]) :- sort_list(L1,L2),
                          sort_list2(T1,T2).
```

Fig. 4.25: The implementation of rule $\Box'$.

The implementation of the $\Box'$ rule is shown in Figure 4.25.

Now we will see in some detail the three parts of the rule. The implementation of the first part, locally_reduced/1 is shown in Figure 4.26. The predicate succeeds if the parameter L is a sequent in NF2 which is locally reduced and has at least one term of the form k(A,X).

```
1.  locally_reduced([])                 :-  member(k(_,_),L),
                                            quasi_locally_reduced(L).

2.  quasi_locally_reduced([],[]).
3.  quasi_locally_reduced([H|T])        :-  quasi_lr(H), quasi_locally_reduced(T).

4.  quasi_lr(p(_,_)).
5.  quasi_lr(k(_,_)).
6.  quasi_lr(plit(_)).
7.  quasi_lr(nlit(_)).
```

Fig. 4.26: The implementation of predicate locally_reduced/1**.**

The call to the predicate locally_reduced/1 just tests whether there is at least one formula of the form k(N,X) in L (clause 1) and all formulæ of L are literals or formulæ of the form k(N,X) or p(N,X) (clauses 2–7.)

The second part of the implementation of the $\Box'$ rule is performed by the predicate get_kp/3. Once the predicate locally_reduced/1 has succeeded

with input L, the rule $\Box'$ may be applied on L (this is the reason of the cut in clause 1 in Figure 4.25; this is a green cut, since no other rule is applicable, and aims at avoiding useless backtracking.) The implementation is shown in Figure 4.27.

```
1.  get_kp([],[],[]).
2.  get_kp([k(N,X)|T],[k(N,X)|T1],L2)   :-  get_kp(T,T1,L2).
3.  get_kp([p(N,X)|T],L1,[p(N,X)|T2])   :-  get_kp(T,L1,T2).
4.  get_kp([plit(_)|T],L1,L2])          :-  get_kp(T,L1,L2).
5.  get_kp([nlit(_)|T],L1,L2])          :-  get_kp(T,L1,L2).
```

Fig. 4.27: The implementation of predicate get_kp/3**.**

This predicate instantiates L2 with a list of all formulæ of the form k(A,X) contained in L and L3 with a list of all formulæ of the form p(A,X) contained in L.

The third part is performed by the predicate get_premises(L1,L2,Lx), where L1 and L2 are the lists of all formulæ of the form k(A,X) and p(A,X) respectively, obtained from predicate get_kp/3 and Lx is a list of terms of the form ap(A,P), where A is an agent and P is a premise. The implementation of this rule is shown in Figure 4.28.

```
1.  get_premises([],_,[]).
2.  get_premises([k(N,X)|T1],L,[ap(N,[X|T2])|T3]) :- get_p(N,L,T2),
                        get_premises(T1,L,T3).

4.  get_p(_,[],[]).
5.  get_p(N,[p(N,X)|T1],[X|T2])   :-  !, get_p(N,T1,T2).
6.  get_p(N,[_|T],L)              :-  !, get_p(N,T,L).
```

Fig. 4.28: The implementation of predicate get_premises/5**.**

The procedure is simple: for each formula k(A,X) in list L1, a list [X|T]

must be constructed, where T is the collection of all formulæ Y such that p(A,Y) is in L2. This is done by predicate get_p(A,L2,T), where L2 is the list obtained from predicate get_kp. The variable T is instantiated to the collection of formulæ described above.

The remaining rules interact with annotations. We begin with rule rep′:

$$\mathsf{rep}' \ \frac{}{\Gamma,\ \boxplus_{[H|\Gamma]}\varphi\ :\ l}$$

In this case, it is necessary to check whether the context $\Gamma : l$ is in the annotation.

The predicate rep_rule/2 does not require much explanation. The input is the list L containing the sequent we want to test for repetition and the list A containing its annotation. The context is obtained with the predicate get_uset/2, which just eliminates the annotated formula from L. Finally, the resulting contextis checked for membership in the annotation A.

The implementation is shown in Figure 4.29. Although all clauses have been put together, the get_uset/2 predicate is in the main module and the rule_rep/2 predicate, as all predicates implementing rules, is in the mod_proof module. The cuts are put only to eliminate redundant backtracking.

```
1.  rep_rule(L,A)    :-  get_uset(L,C), member(C,A), !.
2.  get_uset(L,C)    :-  drop(L,a(_),C), sort_list(C1,C).
```

Fig. 4.29: The implementation of rule rep′.

In the case of the rule foc′, we have to look in the priority list which formula is the first one and to change the list afterwards. The implementation of the rule is in predicate foc_rule/4 of module mod_proof. Predicate foc(L1,L2,P1,P2), where L1, P1 and L2, P2 are the sequents and the priority lists before and after the rule has been applied. If there is no focusable formula in L1 the predicate fails. The implementation is in Figure 4.30.

```
1.  foc_rule(_,_,[],_)              :-  !, fail.
2.  foc_rule(L1,L2,[H|T],P)         :-  foc_rule(L1,L1a,H), !,
                                        append(T,[H],P), sort_list(L1a,L2).
3.  foc_rule(L1,L2,[H|T1],[H|T2])   :-  foc_rule(L1,L2,T1,T2).

4.  foc_rule([],_,_)                :-  !, fail.
5.  foc_rule([c(X)|T],[a(X)|T],c(X)).
6.  foc_rule([H|T1],[H|T2],F)       :-  foc_rule(T1,T2,F).
```

Fig. 4.30: The implementation of rule rep′.

The cuts in clauses 1 and 4 are to avoid useless backtracking. These are green cuts. The cut in clause 2 is a red cut to rule out redundant backtracking. This cut commits the program to the first solution found. When predicate foc_rule(L1,L2,P1,P2) is called, for each formula F in the annotation P1, the predicate foc_rule(L1,L1a,F) is called. If F is in L1, then clause 5 succeeds, the list is rearranged, and clause 2 succeeds. Otherwise, when all formulæ of L1 have been searched (clause 6), clause 4 fails and a new formula of P1 is tested (clause 3). If none of the formulæ of P1 is in L1, the predicate fails (clause 1.)

The last implementation of a rule to be analysed is that of $\boxtimes'_H$. Not surprisingly, its implementation is very similar to that of $\boxtimes'$. It is shown in Figure 4.31.

The a_unfold/3 predicate is entirely analogous to the predicate c_rule/3, seen above (see Figure 4.24.)

4.9 Implementation of the Decision Procedure

The main part of the program is the decision procedure, which is a straightforward implementation of Algorithm 3.42 of Chapter 3. We will explain here a simplified version of the procedure, where the list of parameters has been reduced to its essential elements, i.e., all the arguments needed for

```
1.  a_rule(L1,L2,L3)                    :-  a_rule1(L1,L2a,L3a),
                                            no_dup(L2a,L2b), no_dup(L3a,L3b),
                                            sort_list(L2b,L2), sort_list(L3b,L3).

2.  a_rule1([],_,_)                     :-  !, fail.
3.  a_rule1([a(X)|T],[E1|T],[E2|T])     :-  a_unfold(X,E1,E2).
4.  a_rule1([H|T1],[H|T2],[H|T3])       :-  a_rule1(T1,T2,T3).

5.  a_unfold(X,E1,E2)                   :-  agents(L), a_unfold(X,E1,E2,L).

6.  a_unfold(X,k(N,X),k(N,a(X)),[N]).
7.  a_unfold(X,y(k(N,X),Z1),y(k(N,a(X)),Z2),[N|T]) :- a_unfold(X,Z1,Z2,T).
```

Fig. 4.31: The implementation of rule $\boxtimes'_H$.

the logs of the proof will be omitted. The complete listing may be found in Appendix A.

The decision procedure begins by calling the predicate proof_search(L,R), where L is the sequent in NF2 and R is a variable with will be instantiated with 0 if the proof is successful or 1 if it is not.

The first part is shown in Figure 4.32.

The predicate get_ck(L,P) extracts all occurrences of subformulæ of the form c(X) of L and collects them in P. It is used to construct the priority list of the formulæ to be focused.

The call to the predicate proof_search(L,P,A,K,R) starts the actual decision procedure. The parameters are the following ones:

- L is the sequent whose validity we want to establish.
- P is the priority list.
- A is the annotation, expressed as a list of lists of formulæ.
- K is the code of the current node. It is a natural number.

```
1.  proof_search(L,R)              :-  get_ck(L,P), seq2tring(L,S),
                                       proof_search(L,S,P,[],[],'',0,R,0).
2.  get_ck([],[]).
3.  get_ck([plit(_)|T],L)          :-  get_ck(T,L).
4.  get_ck([nlit(_)|T],L)          :-  get_ck(T,L).
5.  get_ck([(X V Y)|T],L)          :-  get_ck([X,Y|T],L).
6.  get_ck([(X & Y)|T],L)          :-  get_ck([X,Y|T],L).
7.  get_ck([k(_,X)|T],L)           :-  get_ck([X|T],L).
8.  get_ck([p(_,X)|T],L)           :-  get_ck([X|T],L).
9.  get_ck([u(X)|T],L)             :-  get_ck([X|T],L).
10. get_ck([c(X)|T1],[c(X)|T2])    :-  get_ck([X|T1],T2).
```

Fig. 4.32: The implementation of the decision procedure, part I.

- R is a variable that will be instantiated to 0 is the sequent is valid and to 1 otherwise.

The actual implementation uses more parameters to produce the final output, as shown in Section 4.5. From the point of view of the implementation of the proof system, the only parameters that are strictly necessary are these ones. In addition to this, to implement Algorithm 3.42, we need a way to determine cyclicity. It is thus necessary to store some extra information, what is done in the tables history/3 and parent/2. The former table has infomation of nodes that are conclusions of the $\Box'$ rules (recall that in Algorithm 3.42 cyclicity is tested only for such nodes) and the latter stores the topography of the tree.

The three parameters of the table history/3 are (L,A,K), where

- L is sequent in NF2 without literals and annotated formulæ.
- A is the annotation in NF2.
- K is the code of the node, as before.

The implementation of the test for cyclicity is shown in Figure 4.33.

```
1.  is_cyclic(L,P,K0,K)                 :-  get_context(L,C), history(C,P,K0),
                                            ancestor(K0,K).
2.  ancestor(K0,K1)                     :-  parent(K0,K1).
3.  ancestor(K0,K1)                     :-  parent(K1a,K1), ancestor(K0,K1a).
4.  get_context([],[])                  :-  !.
5.  get_context([a(X)|T],[c(X)|T])      :-  !.
6.  get_context([H|T1],[H|T2])          :-  !, get_context(T1,T2).
```

Fig. 4.33: The implementation of the test for cyclicity.

The simplified code of the decision procedure is shown in Figure 4.34. The complete source code can be found in Appendix A.

The last clause (clause 12) succeeds only when all others have failed. It succeeds with irreducible nodes. In the case of the premises of rule $\Box'$, a disjunctive proof process is started. The process is shown in Figure 4.35

Once the list of all possible premises have been constructed with predicates k_rule/3 and get_codes/5, predicate or_proof_search(Lx,P,A,Kx,1,R) is called. Here Lx is a list of all premises, P is the priority list, A is the annotation (if any), Kx is a list of terms of the form ap(A,K) where A is an agent (which is not used in this predicate) and K is the code of an A-premise. R gets instantiated to 0 if some sequent of the list Lx is proved and to 1 otherwise. The fifth predicate has the current value of R. The initial value is 1. This predicate calls proof_search/5 for each sequent of the list Lx.

Some explanation about the encoding of the nodes is in order. In the case of binary trees, it is enough to have a binary word: the code $\langle\rangle$ is the code of the root, the two child nodes are $\langle 0\rangle$ and $\langle 1\rangle$, the two child nodes of $\langle 0\rangle$ are $\langle 00\rangle$ and $\langle 01\rangle$, and so on. A binary word identifies uniquely a node [5]. The proof tree of a preproof in S'_{CK} is not binary (because of rule $\Box'$) and this coding does no longer work. To avoid ambiguities, the codes in the proof tree could use a separator (for instance a point) marking each new entry. But then we have the drawback of the length of the codes for deep trees. Therefore we chose just to assign natural numbers to the codes,

```
1.   proof_search(L,_,_,_,0)   :-  id_rule(L), !.
2.   proof_search(L_,A_,0)     :-  rep_rule(L,A), !.
3.   proof_search(L,P,_,K,1)   :-  is_cyclic(L,P,_,K), !.
4.   proof_search(L,P,A,K,R)   :-  o_rule(L,L1), !, get_code(K,0,K0),
                                   proof_search(L1,P,A,K0,R).
5.   proof_search(L,P,A,K,R)   :-  y_rule(L,L1,L2), !,
                                   get_code(K,0,K0), get_code(K,1,K1),
                                   proof_search(L1,P,A,K0,R1),
                                   proof_search(L2,P,A,K0,R2),
                                   R is max(R1,R2).
6.   proof_search(L,P,A,K,R)   :-  u_rule(L,L1), !, get_code(K,0,K0),
                                   proof_search(L1,P,A,K0,R).
7.   proof_search(L,P,[],K,R)  :-  foc_rule(L,L,P,P1), !, get_code(K,0,K0),
                                   proof_search(L1,P1,[],K0,R).
8.   proof_search(L,P,A,K,R)   :-  c_rule(L,L1,L2), !,
                                   get_code(K,0,K0), get_code(K,1,K1),
                                   proof_search(L1,P,A,K0,R1),
                                   proof_search(L2,P,A,K0,R2),
                                   R is max(R1,R2).
9.   proof_search(L,P,A,K,R)   :-  a_rule(L,L1,L2), !,
                                   get_code(K,0,K0), get_code(K,1,K1),
                                   proof_search(L1,P,A,K0,R1),
                                   proof_search(L2,P,A,K0,R2),
                                   R is max(R1,R2).
10.  proof_search(L,P,A,K,R)   :-  k_rule(L,Lxx,N), !, assert(history(L1,P,K)),
                                   get_codes(Lxx,Lx,Kx,K),
                                   or_proof_search(Lx,P,A,Kx,1,R).
11.  proof_search(_,_,_,_,1)   :-  !.
```

Fig. 4.34: The implementation of the decision procedure, part II.

although this has some extra difficulties to determine the ancestor nodes, for instance the need to have the dynamic table parent/2.

The following step is to show that this program is actually an implemention of Algorithm 3.42, reproduced for convenience in Figure 4.36.

```
1.  or_proof_search([],_,_,_,R,R).
2.  or_proof_search([L|Lx],P,A,[ap(_,K)|Kx],R0,R) :- focused(L), !,
        proof_search(L,P,A,K,R1),
        R2 is min(R0,R1),
        or_proof_search(Lx,P,A,Kx,R2,R).
3.  or_proof_search([L|Lx],P,A,[ap(_,K)|Kx],R0,R) :- !,
        proof_search(L,P,[],K,R1), R2 is min(R0,R1),
        or_proof_search(Lx,P,A,Kx,R2,R).

4.  focused(L) :- member(a(_),L).
```

Fig. 4.35: The implementation of the decision procedure, part III.

Recall that a non-terminal node is one that is not cyclic and not irreducible. The proof is for the simplified version. The extension of the proof to the full version is straightforward, although rather tedious.

Proposition 4.7 *The predicate* proof_search(L,R) *is an implementation of Algorithm 3.42. If at the end* R *is instantiated to 0, the execution is successful; if it is instantiated to 1, it is failed.*

Proof. It is not hard to see that as a result of the execution of predicate get_ck/2, the list P is a list containing all subformulæ of the form c(X) in L (see Figure 4.32.)
Observe that each instance of proof_search/5 corresponds to a branch of the preproof; when a node branches out (for instance due to an application of $\wedge'$), two calls to proof_search/5 are done, one for each new branch. The same happens in the rule $\Box'$. Besides, any terminal node closes the branch, because either clause 3 or clause 11 (Figure 4.34) succeeds with R = 1. Otherwise, the program will try to apply the following rules in this order: id′, rep′, $\vee'$, $\wedge'$, $\diamondsuit'$, foc′, $\circledast'$, $\boxdot'_H$, $\Box'$.
As long as a rule of the set $\{\mathsf{id}', \mathsf{rep}', \vee', \wedge', \diamondsuit'\}$ is applicable, it will be applied (clauses 1, 2 and 4–6 of Figure 4.34.) Besides, because of the cuts, no other rule will be applied in backtracking: the program is commited to the

```
1.  input:  a history-free sequent Γ;
2.  output: a preproof 𝒟 of Γ : l in S'_CK;
3.  begin
4.          set 𝒟 := Γ : l(Γ);
5.          while
6.                there are non-terminal leaves in 𝒟
7.          do
8.                apply id', rep', ∨', ∧', ⋄'
9                     to non-terminal leaves
10.                   until no longer possible;
11.               apply foc' to non-terminal leaves
12                    until no longer possible;
13.               apply id', rep', ∨', ∧', ⋄', ⊡
14                    to non-terminal leaves
15.                   until no longer possible;
16.               apply ⊡'_H to non-terminal leaves
17.                   until no longer possible;
18.               apply ∧' to non-terminal leaves
19.                   until no longer possible;
20.               apply □' to non-terminal leaves
21.                   until no longer possible;
22.         od;
23. end.
```

Fig. 4.36: Algorithm to construct a preproof in CK.

first solution it finds. This represents the loop of lines 8–10 (Figure 4.36.) If after that foc' is applicable, then it will be applied (clause 7 of Figure 4.34.) In the same way, as long as a rule of $\{\mathsf{id}', \mathsf{rep}', \vee', \wedge', \diamondsuit', \boxtimes'\}$ is applicable, it will be applied (clauses 1, 2, 4–6 and 8 of Figure 4.34; this represents the loop of lines 13–15 of Figure 4.36.) Rule $\boxtimes'_H$ is only applied when none of the others is (clause 9 of Figure 4.34.)

As explained in Chapter 3, after the application of rule $\boxtimes'_H$, it is possible that several instances of rule $\wedge'$ are necessary. Finally, when no other rule is applicable, $\Box'$ is applied (clause 10 of Figure 4.34.)

Now we show that if the variable R is instantiated to 0 the execution is successful and if it is instantiated to 1, the execution is failed. This we do by induction on the number of recursive calls to proof_search/5.

Base case: if there is one call to proof_search/5, then it must be the case that clause 1, 2, 3, or 11 succeed, since all others call recursively the predicate. If clause 1 or 2 succeeds, then we have either an instance of id′ or of rep′ and therefore a successful node and in both cases R is instantiated to 0. If clauses 3 or 11 succeed we have a cyclic node or an irreducible one and thus an unsuccessful node. The variable R is instantiated to 1.

Induction step: assume we have $q + 1$ calls to proof_search/5.

Rules $\vee'$, $\diamondsuit'$ and foc′: if any of these rules is applied after a call to predicate proof_search(L,P,A,K,R), a recursive call proof_search(L1,P1,A,K0,R) follows. By induction hypothesis, if this node is successful, R will be instantiated to 0 and otherwise it will be instantiated to 1.

Rules $\wedge'$, $\boxtimes'$, and $\boxtimes'_H$: if any of these rules is applied after a call to proof_search(L,P,A,K,R), two recursive calls proof_search(L1,P,A1,K0,R1) and proof_search(L2,P,A2,K1,R2) respectively follow. By induction hypothesis, the variables R1 and R2 will be instantiated to 0 or 1 depending on whether the respective nodes are successful or unsuccessful. Since R is instantiated to the maximum value of R1 and R2, if both premises are successful R will be instantiated to 0 and it will be instantiated to 1 otherwise.

Rule $\Box'$: if this rule is applied after a call to proof_search(L,P,A,K,R), this yields a call to the predicate or_proof_search(Lx,P,A,Kx,1,R) where Lx is a list of all premises and Kx is a list of terms of the form ap(Ag,Kn), where Ag is an agent and Kn is the code of a Ag-premise.

Observe that the rule or_proof_search/5 is called with an initial value of the comparison parameter set to 1 (the fourth parameter, see Figure 4.35.) There is a call to proof_search/5 for each one of the sequents of the list of

the first parameter (clause 13 of Figure 4.35) and the resulting value of R1 is compared to the current comparison value. If any of the premises in the list of the first parameter succeeds, the value of the comparison parameter is set to 0 and remains in this value. If none of the calls to proof_search/5 succeeds, the value of the comparison parameter remains in 1. When the list is empty, the value of R is set to the current comparison value. ■

Chapter 5

Some Examples

5.1 Introduction

In this chapter we show several examples of the use of the program. The examples range from simple ones to others that are more complex. For typographic reasons, we have replaced in all outputs the symbol ¬ by the symbol ˜.

Section 5.2 contains the examples. In Section 5.3 we make some comparisons on the performance of the program in different conditions: for instance, the same sequent under different number of agents, the same sequent with irrelevant formulæ, or valid sequents with the $\boxtimes$ operator preceding the corresponding formula of a valid sequent. Section 5.4 contains some conclusions.

5.2 Examples

We begin with two very simple examples to give a flavour of the way the program works and the outputs it produces.

Example 5.1 We prove that the sequent $\{(p \vee q), (\neg p \wedge \neg q)\}$ is valid. The goal prove('(p V q), (¬q & ¬p)',ex01) stores the following output onto file ex01.txt:

```
** Agents  [1, 2]
** Sequent [(p V q),(~q & ~p)]

Formula Nr.1: (p V q)
OK

Formula Nr.2: (~q & ~p)
OK

** Parsing OK. Proof-search goes on.

Node <0> = [(p V q), (~q & ~p)]
OR rule yields node <1>

Node <1> = [p, q, (~q & ~p)]
AND rule yields nodes <2> and <3>

Node <2> = [~q, p, q]
instance of ID -- Node successful

Node <3> = [~p, p, q]
instance of ID -- Node successful

** Total number of nodes      = 4
** Maximum height of the tree = 3
** Time elapsed               = 945e-06 sec

** The sequent is valid.
```

Recall that the height of the tree is measured in number of nodes. The value 3 for the height means that there are three nodes from the root to the highest leaf, both included. ♠

Example 5.2 Consider $\Gamma = \{p, (\neg p \wedge \neg q)\}$, which is not valid. The output file has the following contents:

```
** Agents  [1, 2]
** Sequent [p, (~p & ~q)]

Formula Nr.1: p
OK
```

```
Formula Nr.2:  (~p & ~q)
OK

** Parsing OK. Proof-search goes on.

Node <0> = [p, (~p & ~q)]
AND rule yields nodes <1> and <2>

Node <1> = [~p, p]
instance of ID -- Node successful

Node <2> = [~q, p]
irreducible node -- Node unsuccessful

** countermodel: M=(S,R,V)

S = {s0}

R = {R1,R2}
  R1= {}
  R2= {}

V = {(s0,[q])}

** Total number of nodes      = 3
** Maximum height of the tree = 2
** Time elapsed               = 875e-06 sec

** The sequent is not valid.
```

♠

Example 5.3 We consider the sequent $\{\Box_1\neg p, \Diamond\Box_1\neg p\}$ with one single agent.

```
** Agents  [1]
** Sequent [K1~p, UK1~p]

Formula Nr.1: K1~p
OK
```

```
Formula Nr.2:  UK1~p
OK

** Parsing OK. Proof-search goes on.

Node <0> = [K1~p, UK1~p]
U rule yields node <1>

Node <1> = [K1~p, P1UK1~p, P1K1~p]
BOX rule yields node <2>

Node <2> = [~p, UK1~p, K1~p]
U rule yields node <3>

Node <3> = [~p, K1~p, P1UK1~p, P1K1~p]
BOX rule yields node <4>

Node <4> = [~p, UK1~p, K1~p]
U rule yields node <5>

Node <5> 0 [~p, K1~p, P1UK1~p, P1K1~p]
cyclic node: twin node is <3> -- Node unsuccessful

** countermodel: M=(S,R,V)

S = {s0,s1,s2}

R = {R1}
  R1= {(s0,s1),(s1,s2),(s2,s2)}

V = {(s0,[]),
     (s1,[p]),
     (s2,[p])}

** Total number of nodes      = 6
** Maximum height of the tree = 6
** Time elapsed               = 1707e-06 sec

** The sequent is not valid.
```

Example 5.4 We try to prove the same sequent as in the last example, namely $\{\Box_1\neg p, \Diamond\!\!\!\!-\Box_1\neg p\}$ with two agents.

```
** Agents  [1, 2]
** Sequent [K1~p, UK1~p]

Formula Nr.1: K1~p
OK

Formula Nr.2:  UK1~p
OK

** Parsing OK. Proof-search goes on.

Node <0> = [K1~p, UK1~p]
U rule yields node <1>

Node <1> = [K1~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <2>

Node <2> = [K1~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <3>

Node <3> = [K1~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
BOX rule yields node <4>

Node <4> = [~p, UK1~p, K1~p]
U rule yields node <5>

Node <5> = [~p, K1~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <6>

Node <6> = [~p, K1~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <7>

Node <7> = [~p, K1~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
BOX rule yields node <8>

Node <8> = [~p, UK1~p, K1~p]
U rule yields node <9>
```

```
Node <9> = [~p, K1~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <10>

Node <10> = [~p, K1~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]

OR rule yields node <11>

Node <11> 0 [~p, K1~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
cyclic node: twin node is <7> -- Node unsuccessful

** countermodel: M=(S,R,V)

S = {s0,s1,s2}

R = {R1,R2}
  R1= {(s0,s1),(s1,s2),(s2,s2)}
  R2= {}

V = {(s0,[]),
     (s1,[p]),
     (s2,[p])}

** Total number of nodes      = 12
** Maximum height of the tree = 12
** Time elapsed               = 3098e-06 sec

** The sequent is not valid.
```

♠

Example 5.5 Consider the sequent $\{\Box_1\neg p, \Diamond\Box_1\neg p, \Box_2\neg p, \Diamond\Box_2\neg p\}$ with two agents.

```
** Agents  [1, 2]
** Sequent [K1~p, UK1~p,K2~p,UK2~p]

Formula Nr.1: K1~p
OK

Formula Nr.2:  UK1~p
OK
```

```
Formula Nr.3: K2~p
OK

Formula Nr.4: UK2~p
OK

** Parsing OK. Proof-search goes on.

Node <0> = [K1~p, UK1~p, K2~p, UK2~p]
U rule yields node <1>

Node <1> = [UK2~p, K1~p, K2~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <2>

Node <2> = [UK2~p, K1~p, K2~p, (P1K1~p V P2K1p), P1UK1p, P2UK1p]
OR rule yields node <3>

Node <3> = [UK2~p, K1~p, K2~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
U rule yields node <4>

Node <4> = [K1~p, K2~p, (P1UK2~p V P2UK2~p), (P1K2~p V P2K2~p), P1UK1~p,
P1K1~p, P2UK1~p, P2K1~p]
OR rule yields node <5>

Node <5> = [K1~p, K2~p, (P1K2~p V P2K2~p), P1UK1~p, P1UK2~p, P1K1~p,
P2UK1~p, P2UK2~p, P2K1~p]
OR rule yields node <6>

Node <6> = [K1~p, K2~p, P1UK1~p, P1UK2~p, P1K1~p, P1K2~p, P2UK1~p, P2UK2~p,
P2K1~p, P2K2~p]
BOX rule yields nodes <7>, <8>

Node <7> = [~p, UK1~p, UK2~p, K1~p, K2~p]
U rule yields node <9>

Node <9> = [~p, UK2~p, K1~p, K2~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <10>

Node <10> = [~p, UK2~p, K1~p, K2~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <11>
```

```
Node <11> = [~p, UK2~p, K1~p, K2~p, P1UK1~p, P1K1~~p, P2UK1~p, P2K1~p]
U rule yields node <12>

Node <12> = [~p, K1~p, K2~p, (P1UK2~p V P2UK2~p), (P1K2~p V P2K2~p),
P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
OR rule yields node <13>

Node <13> = [~p, K1~p, K2~p, (P1K2~p V P2K2~p), P1UK1~p, P1UK2~p, P1K1~p,
P2UK1~p, P2UK2~p, P2K1~p]
OR rule yields node <14>

Node <14> = [~p, K1~p, K2~p, P1UK1~p, P1UK2~p, P1K1~p, P1K2~p, P2UK1~p,
P2UK2~p, P2K1~p, P2K2~p]
BOX rule yields nodes <15>, <16>

Node <15> = [~p, UK1~p, UK2~p, K1~p, K2~p]
U rule yields node <17>

Node <17> = [~p, UK2~p, K1~p, K2~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <18>

Node <18> = [~p, UK2~p, K1~p, K2~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <19>

Node <19> = [~p, UK2~p, K1~p, K2~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
U rule yields node <20>

Node <20> = [~p, K1~p, K2~p, (P1UK2~p V P2UK2~p), (P1K2~p V P2K2~p),
P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
OR rule yields node <21>

Node <21> = [~p, K1~p, K2~p, (P1K2~p V P2K2~p), P1UK1~p, P1UK2~p, P1K1~p,
P2UK1~p, P2UK2~p, P2K1~p]
OR rule yields node <22>

Node <22> 0 [~p, K1~p, K2~p, P1UK1~p, P1UK2~p, P1K1~p, P1K2~p, P2UK1~p,
P2UK2~p, P2K1~p, P2K2~p]
cyclic node: twin node is <14> -- Node unsuccessful

Node <16> = [~p, UK1~p, UK2~p, K1~p, K2~p]
U rule yields node <23>
```

```
Node <23> = [~p, UK2~p, K1~p, K2~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <24>

Node <24> = [~p, UK2~p, K1~p, K2~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <25>

Node <25> = [~p, UK2~p, K1~p, K2~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
U rule yields node <26>

Node <26> = [~p, K1~p, K2~p, (P1UK2~p V P2UK2~p), (P1K2~p V P2K2~p),
P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
OR rule yields node <27>

Node <27> = [~p, K1~p, K2~p, (P1K2~p V P2K2~p), P1UK1~p, P1UK2~p, P1K1~p,
P2UK1~p, P2UK2~p, P2K1~p]
OR rule yields node <28>

Node <28> 0 [~p, K1~p, K2~p, P1UK1~p, P1UK2~p, P1K1~p, P1K2~p, P2UK1~p,
P2UK2~p, P2K1~p, P2K2~p]
cyclic node: twin node is <14> -- Node unsuccessful

Node <8> = [~p, UK1~p, UK2~p, K1~p, K2~p]
U rule yields node <29>

Node <29> = [~p, UK2~p, K1~p, K2~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <30>

Node <30> = [~p, UK2~p, K1~p, K2~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <31>

Node <31> = [~p, UK2~p, K1~p, K2~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
U rule yields node <32>

Node <32> = [~p, K1~p, K2~p, (P1UK2~p V P2UK2~p), (P1K2~p V P2K2~p),
P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
OR rule yields node <33>

Node <33> = [~p, K1~p, K2~p, (P1K2~p V P2K2~p), P1UK1~p, P1UK2~p, P1K1~p,
P2UK1~p, P2UK2~p, P2K1~p]
OR rule yields node <34>
```

```
Node <34> = [~p, K1~p, K2~p, P1UK1~p, P1UK2~p, P1K1~p, P1K2~p, P2UK1~p,
P2UK2~p, P2K1~p, P2K2~p]
BOX rule yields nodes <35>, <36>

Node <35> = [~p, UK1~p, UK2~p, K1~p, K2~p]
U rule yields node <37>

Node <37> = [~p, UK2~p, K1~p, K2~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <38>

Node <38> = [~p, UK2~p, K1~p, K2~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <39>

Node <39> = [~p, UK2~p, K1~p, K2~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
U rule yields node <40>

Node <40> = [~p, K1~p, K2~p, (P1UK2~p V P2UK2~p), (P1K2~p V P2K2~p),
P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
OR rule yields node <41>

Node <41> = [~p, K1~p, K2~p, (P1K2~p V P2K2~p), P1UK1~p, P1UK2~p, P1K1~p,
P2UK1~p, P2UK2~p, P2K1~p]
OR rule yields node <42>

Node <42> 0 [~p, K1~p, K2~p, P1UK1~p, P1UK2~p, P1K1~p, P1K2~p, P2UK1~p,
P2UK2~p, P2K1~p, P2K2~p]
cyclic node: twin node is <34> -- Node unsuccessful

Node <36> = [~p, UK1~p, UK2~p, K1~p, K2~p]
U rule yields node <43>

Node <43> = [~p, UK2~p, K1~p, K2~p, (P1UK1~p V P2UK1~p), (P1K1~p V P2K1~p)]
OR rule yields node <44>

Node <44> = [~p, UK2~p, K1~p, K2~p, (P1K1~p V P2K1~p), P1UK1~p, P2UK1~p]
OR rule yields node <45>
```

```
Node <45> = [~p, UK2~p, K1~p, K2~p, P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
U rule yields node <46>

Node <46> = [~p, K1~p, K2~p, (P1UK2~p V P2UK2~p), (P1K2~p V P2K2~p),
P1UK1~p, P1K1~p, P2UK1~p, P2K1~p]
OR rule yields node <47>

Node <47> = [~p, K1~p, K2~p, (P1K2~p V P2K2~p), P1UK1~p, P1UK2~p, P1K1~p,
P2UK1~p, P2UK2~p, P2K1~p]
OR rule yields node <48>

Node <48> 0 [~p, K1~p, K2~p, P1UK1~p, P1UK2~p, P1K1~p, P1K2~p, P2UK1~p,
P2UK2~p, P2K1~p, P2K2~p]
cyclic node: twin node is <34> -- Node unsuccessful

** countermodel: M=(S,R,V)

S = {s0,s1,s2,s3,s4,s5,s6}

R = {R1,R2}
  R1= {(s0,s1),(s4,s5),(s6,s5),(s5,s5),(s1,s2),(s3,s2),(s2,s2)}
  R2= {(s0,s4),(s4,s6),(s6,s6),(s5,s6),(s1,s3),(s3,s3),(s2,s3)}

V = {(s0,[]),
     (s1,[p]),
     (s2,[p]),
     (s3,[p]),
     (s4,[p]),
     (s5,[p]),
     (s6,[p])}

** Total number of nodes      = 49
** Maximum height of the tree = 21
** Time elapsed               = 22335e-06 sec

** The sequent is not valid.
```

♠

In the next example the proof uses annotations.

Example 5.6 Consider the sequent $\{\boxdot((p \vee q) \vee (\neg q \wedge \neg p))\}$ (this is the

sequent of Example 5.2 with the ⊡ operator preceding it.)

```
** Agents  [1, 2]
** Sequent [C((p V q) V (~q & ~p))]

Formula Nr.1: C((p V q) V (~q & ~p))
OK

** Parsing OK. Proof-search goes on.

Node <0> = [C((p V q) V (~q & ~p))]
FOC rule yields node <1>

Node <1> = [C[H]((p V q) V (~q & ~p))]
H = []
C[H] rule yields nodes <2> and <3>

Node <2> = [(K1((p V q) V (~q & ~p)) & K2((p V q) V (~q & ~p)))]
AND rule yields nodes <4> and <5>

Node <4> = [K1((p V q) V (~q & ~p))]
BOX rule yields node <6>

Node <6> = [((p V q) V (~q & ~p))]
OR rule yields node <7>

Node <7> = [(p V q), (~q & ~p)]
OR rule yields node <8>

Node <8> = [p, q, (~q & ~p)]
AND rule yields nodes <9> and <10>

Node <9> = [~q, p, q]
instance of ID -- Node successful

Node <10> = [~p, p, q]
instance of ID -- Node successful

Node <5> = [K2((p V q) V (~q & ~p))]
BOX rule yields node <11>
```

```
Node <11> = [((p V q) V (~q & ~p))]
OR rule yields node <12>

Node <12> = [(p V q), (~q & ~p)]
OR rule yields node <13>

Node <13> = [p, q, (~q & ~p)]
AND rule yields nodes <14> and <15>

Node <14> = [~q, p, q]
instance of ID -- Node successful

Node <15> = [~p, p, q]
instance of ID -- Node successful

Node <3> = [(K1C[H]((p V q) V (~q & ~p)) & K2C[H]((p V q) V (~q & ~p)))]
H = [[]]
AND rule yields nodes <16> and <17>

Node <16> = [K1C[H]((p V q) V (~q & ~p))]
H = [[]]
BOX rule yields node <18>

Node <18> = [C[H]((p V q) V (~q & ~p))]
H = [[]]
instance of REP -- Node successful

Node <17> = [K2C[H]((p V q) V (~q & ~p))]
H = [[]]
BOX rule yields node <19>

Node <19> = [C[H]((p V q) V (~q & ~p))]
H = [[]]
instance of REP -- Node successful

** Total number of nodes      = 20
** Maximum height of the tree = 8
** Time elapsed               = 3145e-06 sec

** The sequent is valid.
```

In the next two examples we use the same sequent, namely $\{\circledast p, \diamondsuit\neg p\}$, first with one agent and then with two.

Example 5.7 We prove the validity of the sequent $\{\circledast p, \diamondsuit\neg p\}$ for the case of one agent.

```
** Agents  [1]
** Sequent [Cp, U~p]

Formula Nr.1: Cp
OK

Formula Nr.2:  U~p
OK

** Parsing OK. Proof-search goes on.

Node <0> = [Cp, U~p]
U rule yields node <1>

Node <1> = [Cp, P1~p, P1U~p]
FOC rule yields node <2>

Node <2> = [C[H]p, P1~p, P1U~p]
H = []
C[H] rule yields nodes <3> and <4>

Node <3> = [K1p, P1~p, P1U~p]
BOX rule yields node <5>

Node <5> = [~p, p, U~p]
instance of ID -- Node successful

Node <4> = [K1C[H]p, P1~p, P1U~p]
H = [[P1~p, P1U~p]]
BOX rule yields node <6>

Node <6> = [C[H]p, ~p, U~p]
H = [[P1~p, P1U~p]]
U rule yields node <7>
```

```
Node <7> = [C[H]p, ~p, P1~p, P1U~p]
H = [[P1~p, P1U~p]]
instance of REP -- Node successful

** Total number of nodes      = 8
** Maximum height of the tree = 6
** Time elapsed               = 1460e-06 sec

** The sequent is valid.
```

♠

Example 5.8 Consider the same sequent of Example 5.2, i.e. $\{\boxdot p, \diamondsuit \neg p\}$, for the case of two agents.

```
** Agents  [1, 2]
** Sequent [Cp, U~p]

Formula Nr.1: Cp
OK

Formula Nr.2:  U~p
OK

** Parsing OK. Proof-search goes on.

Node <0> = [Cp, U~p]
U rule yields node <1>

Node <1> = [Cp, (P1~p V P2~p), (P1U~p V P2U~p)]
OR rule yields node <2>

Node <2> = [Cp, (P1U~p V P2U~p), P1~p, P2~p]
OR rule yields node <3>

Node <3> = [Cp, P1~p, P1U~p, P2~p, P2U~p]
FOC rule yields node <4>

Node <4> = [C[H]p, P1~p, P1U~p, P2~p, P2U~p]
H = []
C[H] rule yields nodes <5> and <6>
```

```
Node <5> = [P1~p, P1U~p, P2~p, P2U~p, (K1p & K2p)]
AND rule yields nodes <7> and <8>

Node <7> = [K1p, P1~p, P1U~p, P2~p, P2U~p]
BOX rule yields node <9>

Node <9> = [~p, p, U~p]
instance of ID -- Node successful

Node <8> = [K2p, P1~p, P1U~p, P2~p, P2U~p]
BOX rule yields node <10>

Node <10> = [~p, p, U~p]
instance of ID -- Node successful

Node <6> = [P1~p, P1U~p, P2~p, P2U~p, (K1C[H]p & K2C[H]p)]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
AND rule yields nodes <11> and <12>

Node <11> = [K1C[H]p, P1~p, P1U~p, P2~p, P2U~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
BOX rule yields node <13>

Node <13> = [C[H]p, ~p, U~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
U rule yields node <14>

Node <14> = [C[H]p, ~p, (P1~p V P2~p), (P1U~p V P2U~p)]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
OR rule yields node <15>

Node <15> = [C[H]p, ~p, (P1U~p V P2U~p), P1~p, P2~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
OR rule yields node <16>

Node <16> = [C[H]p, ~p, P1~p, P1U~p, P2~p, P2U~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
instance of REP -- Node successful
```

```
Node <12> = [K2C[H]p, P1~p, P1U~p, P2~p, P2U~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
BOX rule yields node <17>

Node <17> = [C[H]p, ~p, U~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
U rule yields node <18>

Node <18> = [C[H]p, ~p, (P1~p V P2~p), (P1U~p V P2U~p)]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
OR rule yields node <19>

Node <19> = [C[H]p, ~p, (P1U~p V P2U~p), P1~p, P2~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
OR rule yields node <20>

Node <20> = [C[H]p, ~p, P1~p, P1U~p, P2~p, P2U~p]
H = [[P1~p, P1U~p, P2~p, P2U~p]]
instance of REP -- Node successful

** Total number of nodes      = 21
** Maximum height of the tree = 11
** Time elapsed               = 4451e-06 sec

** The sequent is valid.
```

♠

5.3 Some Comparisons on the Performance

We have run some examples varying the parameters, namely the number of agents. We have that an increment of agents increases linearly the execution time. This is not a serious problem. All the same, the number of agents could be limited to the agents the actually appear in the sequent. We have not done this optimisation.

The following table shows the influence of increasing the number of agents for three different sequents, namely $\{\Box_1\neg p, \Diamond\!\!\!\!-\Box_1\neg p\}$ (used in examples 3 and 4), $\{\boxplus p, \Diamond\!\!\!\!-\neg p\}$ (used in examples 7 and 8) and $\{\boxplus(p\vee q), \Diamond\!\!\!\!-(\neg q\wedge \neg p)\}$. We have the following results:

sequent	agents	size	height	time (μsec)
$\{\Box_1\neg p, \Diamond\!\!\!\!\!-\Box_1\neg p\}$	1	6	6	0
	2	12	12	0
	3	18	18	$1{,}6\times10^3$
$\{\Diamond\!\!\!\!\!-\neg p, \circledast p\}$	1	16	11	$1{,}5\times10^3$
	2	21	15	62×10^3
	3	932	46	$9{,}27\times10^6$
$\{\circledast(p\vee q), \Diamond\!\!\!\!\!-(\neg q\wedge\neg p\}$	1	14	7	821
	2	37	12	$2{,}65\times10^3$
	3	68	17	$7{,}28\times10^3$

In the presence of $\Diamond\!\!\!\!\!-$-formulæ or $\circledast$-formulæ the addition of agents has a linear effect on the height of the tree. In the presence of $\circledast$-formulæ or $\Diamond\!\!\!\!\!-$-formulæ containing branching subformulæ, the number of nodes of the trees increment in a polynomial way. This is acceptable, although in many cases the subtrees are copies of each other and this could be used to optimise the program.

In the next examples we introduce redundant formulæ in the sequents to see their effects on the size and height of the tree. We will work with two agents in all cases and we denote by Γ the sequent $\{\circledast(p\wedge q), \Diamond\!\!\!\!\!-\neg p, \Diamond\!\!\!\!\!-\neg q\}$, to which we will add redundant formulæ. We have:

sequent	size	height	time (msec)
Γ	34	17	3.11
$\Gamma, \Diamond\!\!\!\!\!-v_1, \Diamond\!\!\!\!\!-v_2$	52	29	9.57
$\Gamma, \Diamond\!\!\!\!\!-(v_1\wedge v_2$	73	24	10.64
$\Gamma, \Diamond\!\!\!\!\!-(\circledast v_1\wedge v_2)$	73	24	10.92
$\Gamma, \Diamond\!\!\!\!\!-(\circledast v_1\wedge \circledast v_2)$	73	24	10.92
$\Gamma, ((v_1\vee v_2)\vee v_3)$	88	28	12.30
$\Gamma, \Box_1\Box_2(v_1\wedge v_2)$	154	26	21.58
$\Gamma, ((v_1\wedge v_2)\wedge v_3)$	260	28	60.92
$\Gamma, (v_1\wedge v_2), (v_3\wedge v_4)$	347	28	103.91
$\Gamma, \circledast v_1$	718	29	227.22
$\Gamma, \circledast v_1, \circledast v_2$	4422	31	14278.81

Example 5.9 We perform a proof of the induction axiom $(\Box\varphi\wedge\circledast(\varphi\Rightarrow\Box\varphi))\Rightarrow\circledast\varphi$. The corresponding sequent is $\{\Diamond\neg p, \Diamond\!\!\!\!\!-(p\wedge\Diamond\neg p), \circledast p\}$. The results are shown next.

agents	size	height	time (μsec)
1	10	7	$2{,}75\times10^3$
2	28	14	$10{,}74\times10^3$

♠

We finish with a couple of examples where the performance of the program is less than optimal. This examples were run only for the second version.

Example 5.10 We try to prove the sequents $\Diamond\boxdot\neg p$, $\boxdot\neg p$ and $\Diamond\boxdot(\neg p \wedge q)$, $\boxdot\neg p$. The change of the first formula results in a much worse performance. This was also observed by Schwendimann [78]: branching formulæ worsen the performance.

sequent	size	height	time (msec)
$\Diamond\boxdot\neg p$, $\boxdot\neg p$	226	30	80.41
$\Diamond\boxdot(\neg p \wedge q)$, $\boxdot\neg p$	2792	44	22.54×10^3

♠

5.4 Some Conclusions

We have observed, as noted by Schwendimann [78], that the main problem regarding complexity are the branching rules, especially $\boxdot$, whose premises increase polynomially with the number of agents. There are some possibilities of optimisation. For instance, agents that do not appear in the sequent are not needed in the proof, although they could increase the number of nodes in the tree.

Besides, the possibility of checking whether a branch has already appeared could increase the efficiency of the method, since it is not uncommon that the same branches appear repeatedly.

Chapter 6

Conclusions and Further Work

6.1 Introduction

6.2 A Comparison With Other Approaches

We have found no other implementation of a calculus for common knowledge. Further, we know of no other implementable calculus for common knowledge: some of them have cuts [28, 79], and others have infinitary rules [2, 14, 48]. The only other implementation of common knowledge we are aware of is the tableau system of Abate, Goré and Widmann [1]. It is an adaptation of the system of Schwendimann for PLTL [78]. We describe briefly the main features of this system next.

A *tableau* [35, 78] is a single-rooted finite tree where each node is labelled with a set of formulæ that is derived from the set of formulæ of its parent node. Tableaux systems branch out in instances of disjunctive rules like $\vee$ or $\Diamond$. Thus there is a duality between the trees that represent preproofs in our system and tableaux. We will refer henceforth exclusively to tableaux for CK.

There are several similarities between the tableau approach and ours.

Both are "one-pass" procedures, in the sense that it is not necessary to construct the whole tree and to make a second pass on it to prune useless parts of it as for instance in [27, 59]. The only information which is needed during the construction of the trees is the information of the current branch, in both cases to be able to detect loops. In our case we store the whole tree, but this is not necessary for the decision procedure. It is necessary only if we want to extract a derivation in S_{CK} if the sequent was valid or a countermodel if it was not, as described in Chapter 4.

There are also some important differences. In our case there is a relatively simple proof system underlying the implementation. The nodes of the underlying preproofs, to which the rules are applied, are extended sequents (see Chapter 3, Section 3.5), i.e., sequents together with a priority list. In the case of the tableau system, the rules are applied on tableau nodes, that are rather complex objects, which require some devices like "histories" to spot repetitions and variables, which pass information from children to parents in the tableau. Another difference is that the procedure we follow allows the construction of a countermodel when the sequent is not valid. To extract a countermodel from a tableau system, the whole derivation should be stored. This is not done in the tableau system we comment.

There is a direct correspondence of the rules of the tableau system and the rules of S'_{CK} as shown below.

Tableau	S'_{CK}
id	id'
$\vee$	$\vee'$
$\wedge$	$\wedge'$
[C]	$\boxtimes$
$\langle$C$\rangle$	$\diamondsuit$
$\langle a\rangle$	$\Box'$

Besides, the tableau systems has rules [E] and $\langle$E$\rangle$ for formulæ that correspond to the abbreviations $\Box$ and $\Diamond$ respectively (see Chapter 3, page 25.) The rule $\langle a\rangle$ is an existential branching rule, as is $\Box'$ in S'_{CK}.

A node in the tableau system has the form

$$(\Gamma :: \mathsf{HAg}, \mathsf{HCr} :: \mathsf{mrk}, \mathsf{uev})$$

Where:

- Γ is a set of formulæ.
- HAg is a partial function from formulæ to agents. This function is used in the $\langle \mathrm{E} \rangle$ rule. When this rule is applied to a formula $\Diamond\!\!\!\!-\varphi$, it takes the value of the node that is "tracking" the resulting $\Diamond_i\!\!\!\!-\varphi$ formula.
- HCr is a history containing information concerning the current branch. It contains lists with formulæ of the form $\Diamond_i\varphi$ and $\Box_i\varphi$. New sets of formulæ are added in each instance of $\langle a \rangle$. It is used to detect loops and to "block" formulæ of the form $\Diamond_i\varphi$. A formula of this form is blocked by an ancestor instance of $\langle a \rangle$ if the formula is already active in the ancestor rule instance of $\langle a \rangle$. A blocked formula is not active in another instance of the $\langle a \rangle$ rule.
- mrk is a variable that indicates whether the node is "marked" (closed) or not. A node is closed if (1) it contains complementary literals (i.e., it is inconsistent), (2) it is not an existential node and all its child nodes are closed, or (3) it is an existential node and some of its childs is closed.
- uev ("unfulfilled eventualities") is a partial function mapping formulæ to $\mathbb{N} > 0$. Roughly speaking, if $\mathsf{uev}(\Diamond_i\psi)$ is defined, it means that there is a "loop" starting at the parent node and reaching to a "blocking" node for $\Diamond_i\psi$ higher up in the branch.

Although both methods share several similarities, the set of rules of the tableau method is rather unintuitive and the rôle of the histories and variables is far from clear. In the same way, the blocking mechanism is complicated.

In our case there is a clear difference between the level of the underlying sequent system, where the only syntactic extension on the sequents is the inclusion of the priority lists, and whose rules are relatively simple, and the label of the implementation, where the need of collecting information for the coonstruction of the proof or the countermodel makes the nodes more complicated. This distinction is somewhat blurred in the tableau method, where the set of rules is already very complicated.

The decision problem for CK is known to be EXPTIME-complete [37]. In the tableau-system, worst-case complexity is double exponential on the size of the formula. In our case it is even worse, as shown in Chapter 3, Section 3.6. This is because in the case of the tableau there is a unique control for repetitions. In the annotated system the presence of the priority list is responsible for the increase in worst-case complexity. Here the number of nodes in the worst case is in $\mathcal{O}(2^{(n+1)!*2^n})$, whereas in the tableau approach it is in $\mathcal{O}(2^{2^n})$. The worst case is intractable in both cases.

Many good properties of the tableau approach are also present in ours. One of them, the one-pass nature of both algorithms, has already been mentioned. Another one is the potential for parellisation, either in the form of threads [81] or of separate processes [8].

6.3 Conclusions and Further Work

The proof system presented here is cut-free and finitary. The complexity is in the worst case intractable, but otherwise it is a one-pass tree similar to the tableau method shown in Section 6.2. Some drawbacks of the original proof system [13] are (not unexpectedly) inherited here, notably the lack of a syntactic cut-elimination procedure.

There are some logics to which this approach might work. The first obvious extension is *relativised common knowledge* [9, 79]. The semantics of annotations is very similar to that of the relativised common knowledge. In fact, there is a relativised common knowledge operator which can express the semantics of annotated formulæ. The operator has the form $\boxdot(\varphi, psi)$ and its dual has the form $\diamondsuit(\varphi, \psi)$. Given an epistemic model $\mathcal{M} = (S, R, v)$

and a state $s \in S$, the satisfaction relation for this operator is defined as follows:

$(\mathcal{M}, s) \models \boxed{*}_{[H]}\varphi$ iff for all s-paths s_0, $s_1, \ldots$ in $\mathcal{M}$

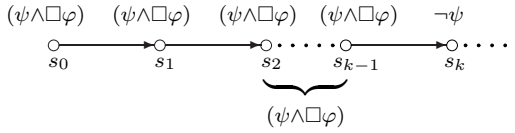
either there is a $k \geq 0$ in the path with $(\mathcal{M}, s_k) \models \neg\psi$
and for all j, $0 \leq j < k$, $(\mathcal{M}, s_j) \models \Box\varphi$,

or $(\mathcal{M}, s_j) \models \Box\varphi$ for all $j \geq 0$.

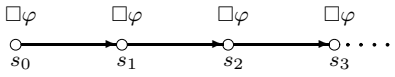
As a consequence, negation of these is operators is defined as follows:

$$\neg\boxed{*}(\varphi, \psi) = \diamondsuit(\varphi, \neg\psi) \quad \neg\diamondsuit(\varphi, \psi) = \boxed{*}(\varphi, \neg\psi)$$

The above definition is equivalent to saying that if $(\mathcal{M}, s) \models \boxed{*}(\varphi, \psi)$, then all s-paths s_0, $s_1, \ldots$ have one of the following shapes:



(the rest does not matter)



(the rest does not change)

Thus it is possible to define a logic of (relativised) common knowledge which is cut-free and finitary without using annotations. The price we pay is that the "usual" common knowledge operator is just a special case of the relativised one.

Another interesting logic is Propositional Dynamic Logic (PDL) [33, 38]. This logic has its roots in the first attempts at formalisation of procedural programming languages [29, 43]. It has a set of atomic programs. More complex programs can be a constructed by means of the atomic ones and the some operations on programs such as concatenation (`;` —semicolon), indeterminate repetition of a program (`*` —asterisk) and random execution of one or another program ($\cup$). It is also possible to construct a program with the *test* operator (`?` —question mark) and a formula A: the program

?A goes on to the next step if A evaluates to *true* and aborts otherwise. Besides the usual propositional formulæ, there are formulæ as $[\alpha]A$ meaning that after execution of program α formula A holds.

Since PDL is a multimodal logic (there is a set of binary relations, one for each atomic program), the method shown here should be applicable. There are non-cut-free axiomatisations for PDL [69] or cut-free axiomatisations for some fragments fo PDL [16].

Another possibility is the investigation of some optimisations for the algorithm used here. Some optimisation research done in other areas, notably Description Logics [44, 45, 46] might be useful to our problem. The performance of the program might also be enhanced by avoiding multiple proofs of branches that are identical up to the names of the propositional variables. In all cases, a more efficient implementation would require the use of a procedural language. Prolog is a nice language for fast prototyping, but the heavy use of recursion puts a heavy burden on the resources of the system.

Appendix A

Source Code

A.1 Introduction

This Appendix contains the whole source code together with the comments that provide succinct explanations on what the different predicates do and what their parameters represent. In order to ease the lecture of the code, some page breaks heve been introduced so as to keep sections of the code together.

The Appendix is organised as follows: Section A.2 contains the code of the main program and all other sections contain the code of the modules used by it: SectionA.3 contains the code of module mod_parse, which performs the parsing of the formulæ of the sequent (the program will not attempt to prove or disprove a syntactically incorrect sequent.) SectionA.4 contains the code of the module mod_proof, which comprises the main predicates to carry over the proof. The code of module mod_services is in Section A.5. This module contains some general-purpose predicates such as list membership, list concatenation, and sorting of terms. The code of module mod_reports is in Section A.6. This module contains the predicates used to write the final log of the proof or the countermodel when the sequent is not a valid one. The negation symbol ($\neg$) has been replaced by the tilde ($\tilde{}$) for typographic reasons.

A.2 The Main Part

The code that follows is the corresponding to the main part of the implementation of S_{CK}.

```
:- use_module(mod_parse).
:- use_module(mod_proof).
:- use_module(mod_services).
:- use_module(mod_reports).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% agents(L) :- L is the list of agents. Only one should be enabled.     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%agents(['1','2','3']).
agents(['1','2']).
%agents(['1']).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% prove(S,X) :- S is the sequent to be proved and X.txt will be the     %%
%%               file that will store the results.                       %%
%% seq2list(L1,L2,L3) :- mod_parse; L1 is NF0, L2 is a list of formulae  %%
%%               in NF0 and L3 a list of formulae in NF1.                %%
%% parse_seq(L1,L2,L3) :- mod_parse; L1 is a list of formulae in NF1     %%
%%               and L2 is the list of formulae in NF2. L3 is a list of  %%
%%               codes of errors (0=OK, 1=syntax error.)                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

prove(S,X) :- get_time(T0), string_concat(X,'.txt',XX), open(XX,write,OO),
        agents(Ag), get_agents(Ag,StAg),
        list_concat(['** Agents  [',StAg,']'],St0), write(OO,St0), nl(OO),
        list_concat(['** Sequent [',S,']'],St), write(OO,St), nl(OO),
        nl(OO), seq2list(S,S1,S2), parse_seq(S2,F,E),
        first_part(S1,E,F,OO,T0).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% first_part(S,E,F,OO) :- S is the list of formulae (NF0), E is the     %%
%%              list of  parse results (0=OK, 1=syntax error), F is the  %%
%%              list of terms (NF2) and OO is the output stream.         %%
%% first_part(S,E,F,OO,N,M) :- S, E, F, and OO as before; N=1 (errors),  %%
%%              N=0 (no errors), and M is a sequence-number for the      %%
%%              formulae.                                                %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

first_part(S,E,F,OO,TO) :- first_part(S,E,F,OO,0,1,TO).
first_part([],[],F,OO,0,_,TO) :-
        write(OO,'** Parsing OK. Proof-search goes on.'), nl(OO), nl(OO),
        proof_search(F,R), last_part(R,OO,TO), forget_all.
first_part([],[],_,OO,1,_,_) :-
        write(OO,'** There were syntax errors. Process aborted.'),
        close(OO).
first_part([S|T1],[0|T2],F,OO,N,M,TO) :-
        list_concat(['Formula Nr.',M,': ',S],St),
        write(OO,St), nl(OO), write(OO,'OK'), nl(OO), nl(OO),
        plus(M,1,M1), first_part(T1,T2,F,OO,N,M1,TO).
first_part([S|T1],[1|T2],F,OO,_,M,TO) :-
        list_concat(['Formula Nr.',M,': ',S],St),
        write(OO,St), nl(OO), write(OO,'Syntax error in formula'),
        nl(OO), nl(OO), plus(M,1,M1), first_part(T1,T2,F,OO,1,M1,TO).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% last_part(R,OO) :- R is the result of proof_search                   %%
%%              (R=0 the sequent is valid, R=1 the sequent is not       %%
%%              valid) and OO is the output stream.                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

last_part(0,OO,TO) :- get_derivation(OO), sum_up(OO,TO),
        write(OO,'** The sequent is valid.'), nl(OO), close(OO).
last_part(1,OO,TO) :- get_preproof(OO), get_model(OO), sum_up(OO,TO),
        write(OO,'** The sequent is not valid.'), nl(OO), close(OO).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% forget_all :- the dynamic databases are deleted.                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

forget_all :- retractall(history(_,_,_)),
        retractall(deriv(_,_,_,_,_,_,_,_,_)),
        retractall(xcode(_)), retractall(dcode(_)),
        retractall(parent(_,_)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% proof_search(L,R) :- L is a sequent (NF2) and proof-search is        %%
%%             performed on it; R=0 means that there was a proof;      %%
%%             R=1 means that the proof-search failed.                 %%
%% proof_search(L,S1,P,A,S2,K,X,R,W,H) :- proof-search for sequent L;   %%
%%             P is the list of priorities, S1 is the sequent in NF0,  %%
%%             A is an annotation (NF2), S2 is the annotation in NF0,  %%
%%             K is the code of the node in the proof-tree,            %%
%%             X=0 (history-free) or X=1 (annotated sequent),          %%
%%             R=0 (successful node), R=1 (unsuccessful node) and      %%
%%             W=0 (no occurrence of K yet), W=1 (at least one         %%
%%             instance of K; H is the height of the proof.            %%
%% deriv(L,S1,S2,K,Rule,X,R,Succ,H) :- P is the list of negative        %%
%%             literals occurring in the sequent; S1, S2, K, and X as  %%
%%             before; 'Rule' is the rule applied to the sequent and   %%
%%             the list 'Succ' has the codes of the successors; H is   %%
%%             the height of the preproof.                             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proof_search(L,R) :- get_ck(L,P), seq2string(L,S), assert(xcode(0)),
        proof_search(L,S,P,[],[],0,0,R,0,1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                        %%
%%   Decision Procedure   %%
%%                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

proof_search(L,S1,_,_,S2,K,X,0,_,H) :- id_rule(L), !, get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'ID',X,0,[],H)).
```

```
proof_search(L,S1,_,A,S2,K,1,0,_,H) :- rep_rule(L,A), !,
        get_prop(L,Prop), asserta(deriv(Prop,S1,S2,K,'REP',1,0,[],H)).

proof_search(L,S1,P,_,S2,K,X,1,1,H) :- locally_reduced(L),
        is_cyclic(L,P,K0,K), !, get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'o',X,1,[K0],H)).

proof_search(L,S1,P,A,S2,K,X,R,W,H) :- o_rule(L,L1), !,
        seq2string(L1,S1a), plus(H,1,H1), get_xcode(K1),
        assert(parent(K,K1)),
        proof_search(L1,S1a,P,A,S2,K1,X,R,W,H1), get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'OR',X,R,[K1],H)).

proof_search(L,S1,P,A,S2,K,X,R,W,H) :- y_rule(L,L1a,L1b), !,
        seq2string(L1a,S1a), seq2string(L1b,S1b), plus(H,1,H1),
        get_xcode(K1), get_xcode(K2),
        assert(parent(K,K1)), assert(parent(K,K2)),
        proof_search(L1a,S1a,P,A,S2,K1,X,R1,W,H1),
        proof_search(L1b,S1b,P,A,S2,K2,X,R2,W,H1),
        R is max(R1,R2), get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'AND',X,R,[K1,K2],H)).

proof_search(L,S1,P,A,S2,K,X,R,W,H) :- u_rule(L,L1), !,
        seq2string(L1,S1a), get_xcode(K1),
        assert(parent(K,K1)), plus(H,1,H1),
        proof_search(L1,S1a,P,A,S2,K1,X,R,W,H1), get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'U',X,R,[K1],H)).

proof_search(L,S1,P,[],[],K,0,R,W,H) :- foc_rule(L,L1,P,P1), !,
        seq2string(L1,S1a), get_xcode(K1),
        assert(parent(K,K1)), plus(H,1,H1),
        proof_search(L1,S1a,P1,[],[],K1,1,R,W,H1),
        asserta(deriv(L,S1,[],K,'FOC',0,R,[K1],H)).
```

```
proof_search(L,S1,P,A,S2,K,X,R,W,H) :- c_rule(L,L1a,L1b), !,
        seq2string(L1a,S1a), seq2string(L1b,S1b),
        get_xcode(K1), get_xcode(K2), plus(H,1,H1),
        assert(parent(K,K1)), assert(parent(K,K2)),
        proof_search(L1a,S1a,P,A,S2,K1,X,R1,W,H1),
        proof_search(L1b,S1b,P,A,S2,K2,X,R2,W,H1),
        R is max(R1,R2), get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'C',X,R,[K1,K2],H)).

proof_search(L,S1,P,A,S2,K,1,R,W,H) :- a_rule(L,L1a,L1b), !,
        get_context(L,C), seq2string(L1a,S1a), seq2string(L1b,S1b),
        seq2string(C,C1), get_xcode(K1), get_xcode(K2),
        assert(parent(K,K1)), assert(parent(K,K2)), plus(H,1,H1),
        proof_search(L1a,S1a,P,[],[],K1,0,R1,W,H1),
        proof_search(L1b,S1b,P,[C|A],[C1|S2],K2,1,R2,W,H1),
        R is max(R1,R2), get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'C[H]',1,R,[K1,K2],H)).

proof_search(L,S1,P,A,S2,K,X,R,_,H) :- k_rule(L,Lxx), !,
        get_context(L,C), assert(history(C,P,K)),
        get_codes(Lxx,Lx,Kx,K),
        or_proof_search(Lx,P,A,S2,Kx,1,R,H), get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'BOX',X,R,Kx,H)).

proof_search(L,S1,_,_,S2,K,X,1,_,H) :- !, get_prop(L,Prop),
        asserta(deriv(Prop,S1,S2,K,'i',X,1,[],H)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% or_proof_search(Lx,P,A,S2,Kx,R1,R,H) :- Lx is the list of premises   %%
%%              of rule K, P, A and S2 as before, Kx is a list of       %%
%%              terms ap(A,Kn) where A is an agent and Kn the code an   %%
%%              A-premise, R1 is an accumulator initially 1, R is the   %%
%%              result and H is the height of the node.                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

or_proof_search([],_,_,_,_,R,R,_).
or_proof_search([L|Lx],P,A,S2,[ap(_,K)|Kx],R0,R,H) :- focused(L), !,
        seq2string(L,S1),
        plus(H,1,H1), proof_search(L,S1,P,A,S2,K,1,R1,1,H1),
        R2 is min(R0,R1), or_proof_search(Lx,P,A,S2,Kx,R2,R,H).
```

```
or_proof_search([L|Lx],P,A,S2,[ap(_,K)|Kx],R0,R,H) :- !,
        seq2string(L,S1), plus(H,1,H1),
        proof_search(L,S1,P,[],[],K,0,R1,1,H1),
        R2 is min(R0,R1), or_proof_search(Lx,P,A,S2,Kx,R2,R,H).

focused(L) :- member(a(_),L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_ck(L1,L2) :- L2 is the list of all subformulae of the form       %%
%%              c(A,X) occurring in L1.                                  %%
%% get_xcode(K1)  :- K1 is the next acailable code.                     %%
%% get_codes(Lxx,Lx,Kx,K) :-Lxx is a list of elements of the form       %%
%%              ap(Ag,P)where Ag is an agent and P is a premise by       %%
%%              rule K; Lx is the list of premises and Kx is a list of  %%
%%              elements of the form ap(Ag,Knxt) where Ag is an agent   %%
%%              and Knxt is the code of the Ag-successor.               %%
%% get_context(L,C) :- L is a sequent and C is the corresponding        %%
%%              underlying set.                                         %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_ck([],[])               :- !.
get_ck([plit(_)|T],L)       :- !, get_ck(T,L).
get_ck([nlit(_)|T],L)       :- !, get_ck(T,L).
get_ck([o(X,Y)|T],L)        :- !, get_ck([X,Y|T],L).
get_ck([y(X,Y)|T],L)        :- !, get_ck([X,Y|T],L).
get_ck([k(_,X)|T],L)        :- !, get_ck([X|T],L).
get_ck([p(_,X)|T],L)        :- !, get_ck([X|T],L).
get_ck([u(X)|T],L)          :- !, get_ck([X|T],L).
get_ck([c(X)|T1],[c(X)|T2]) :- !, get_ck([X|T1],T2).

get_xcode(C) :- xcode(N), plus(N,1,C), retract(xcode(N)),
        assert(xcode(C)).

get_codes([],[],[],_).
get_codes([ap(Ag,P)|T1],[P|T2],[ap(Ag,Knxt)|T3],K) :- get_xcode(Knxt),
        assert(parent(K,Knxt)), get_codes(T1,T2,T3,K).

get_context([],[]) :- !.
get_context([a(X)|T],[c(X)|T])  :- !.
get_context([H|T1],[H|T2])  :- !, get_context(T1,T2).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% is_cyclic(L,P,K0,K) :- sequent L (eventually with priority list P)   %%
%%              is cyclic, i.e., there is an entry (L0,P,K) in the      %%
%%              database 'history' such that L0 contains the same       %%
%%              formulae as L and K0 is in the same branch as K.        %%
%% ancestor(K1,K2) :- K1 is lower in the same branch of deriv/8 as K2.  %%
%% parent(K1,K2) :- K1 is the parent node of K2.                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

is_cyclic(L,P,K0,K) :- get_context(L,C), history(C,P,K0), ancestor(K0,K).

ancestor(K0,K1) :- parent(K0,K1).
ancestor(K0,K1) :- parent(K1a,K1), ancestor(K0,K1a).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_prop(L,P) :- the list P contains all propositions p such that    %%
%%              nlit(p) is in sequent L (NF2).                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_prop([],[]).
get_prop([nlit(X)|T1],[X|T2]) :- !, get_prop(T1,T2).
get_prop([_|T],P)             :- !, get_prop(T,P).
```

A.3 The Parsing Module

The mod_parse module implements the code for parsing sequents and formulæ and for conversion among the different representations.

```
:- module(parse,[seq2list/3,parse_seq/3,seq2string/2]).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% seq2list(S,L1,L2) :- The string S (sequent, NF0) yields L1 (the     %%
%%              individual formulae, NF0) and L2 is a list with         %%
%%              contains the symbols without spaces.                    %%
%% Example: if S='(p V q), ((p & (q V ~q)), p' then                      %%
%%          L1=['(p V q)', '((p & (q V ~q))', 'p'] and                   %%
%%          L2=[['(','p','V','q',')'],                                  %%
%%              ['(','(','p','&','q','V','~','q'')',')'],               %%
%%              ['p']]                                                  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
seq2list(S,L1,L2) :- seq_sep(S,L1), seq2list2(L1,L2).
seq2list2([],[]).
seq2list2([H1|T1],[H2|T2]) :- formula2list(H1,H2),
        seq2list2(T1,T2).

formula2list(S,L) :- string_to_list(S,L1), decode(L1,L).

decode([],[]).
decode([32|T],L)        :- !, decode(T,L).
decode([H1|T1],[H2|T2]) :- !, char_code(H2,H1), decode(T1,T2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% seq_sep(S,L) :- S is a string and L is a list containing the       %%
%%              substrings of S that are separated by commas.         %%
%% Example: if S='aa,b,ccc,d', then L=['aa','b','ccc','d']            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

seq_sep(S,[S])    :- first_comma(S,-1), !.
seq_sep(S1,[H|T]) :- !, first_comma(S1,N1), sub_string(S1,0,N1,N2,H),
        plus(N1,1,N1a), plus(N2a,1,N2),
        sub_string(S1,N1a,N2a,_,S2), seq_sep(S2,T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% first_comma(S,N) :- the first comma of S appears at position N.    %%
%%              If S contains no commas, N is -1.                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

first_comma(S,N)     :- first_comma(S,0,N).
first_comma(S,N,-1)  :- string_length(S,N), !.
first_comma(S,N1,N2) :- sub_string(S,N1,1,_,','), !, N2 is N1.
first_comma(S,N1,N2) :- !, plus(N1,1,N1a), first_comma(S,N1a,N2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% parse_seq(L,F,E) :- formulae of sequent L are parsed. The results  %%
%%              are stored in F and the error codes in E.             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

parse_seq([],[],[]).
parse_seq([H|T1],[F|T2],[E|T3]) :- parse_formula(H,F,E),
        parse_seq(T1,T2,T3).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% parse_formula(L,E,N2) :- the formula L (NF1) is parsed yielding    %%
%%              F (NF2) and error code E (E=0: no syntax errors;      %%
%%              E=1: syntax error.)                                   %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

parse_formula([X],plit(X),0)   :- is_prop(X), !.
parse_formula([_],'',1).
parse_formula(['~',X],nlit(X),0) :- is_prop(X), !.
parse_formula(['~',_],'',1).
parse_formula(['C'|T],c(X),N) :- !, parse_formula(T,X,N).
parse_formula(['U'|T],u(X),N) :- !, parse_formula(T,X,N).
parse_formula(['K',M|T],k(M,X),N) :- is_agent(M), !,
        parse_formula(T,X,N).
parse_formula(['K'|_],'',1).
parse_formula(['P',M|T],p(M,X),N) :- is_agent(M), !,
        parse_formula(T,X,N).
parse_formula(['P'|_],'',1).
parse_formula(['('|T],y(X1,X2),N) :- main_conn(T,T1,T2,'&'), !,
        parse_formula(T1,X1,N1), parse_formula(T2,X2,N2),
        N is max(N1,N2).
parse_formula(['('|T],o(X1,X2),N) :- main_conn(T,T1,T2,'V'), !,
        parse_formula(T1,X1,N1), parse_formula(T2,X2,N2),
        N is max(N1,N2).
parse_formula(_,'',1).

main_conn(L1,L2,L3,C) :- main_conn(L1,L2,L3,C,0).

main_conn([],_,_,_,_)           :- !, fail.
main_conn(['&'|T1],[],T2,'&',0) :- append(T2,[')'],T1), !.
main_conn(['V'|T1],[],T2,'V',0) :- append(T2,[')'],T1), !.
main_conn(['('|T1],['('|T2],L3,C,N) :- !, plus(N,1,N1),
        main_conn(T1,T2,L3,C,N1).
main_conn([')'|T1],[')'|T2],L3,C,N) :- !, plus(N1,1,N),
        main_conn(T1,T2,L3,C,N1).
main_conn([H|T1],[H|T2],L3,C,N) :- >=(N,0), !,
        main_conn(T1,T2,L3,C,N).

is_prop(X) :- char_code(X,N), >=(N,97), =<(N,122).
is_agent(X) :- agents(L), member(X,L).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% seq2string(L,S) :- sequent L (NF2) is converted into string S.      %%
%% term2string(F,S) :- formula F (NF2) is converted                     %%
%%               into string S.                                         %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

seq2string(L,S)        :- seq2string(L,'',S).
seq2string([],S,S).
seq2string([H|T],'',S) :- !, term2string(H,S1),
        seq2string(T,S1,S).
seq2string([H|T],S1,S) :- term2string(H,S2),
        list_concat([S1,', ',S2], S3), seq2string(T,S3,S).

term2string(plit(X),X).
term2string(nlit(X),Y) :- string_concat('~',X,Y).
term2string(o(X,Y),S)  :- term2string(X,S1),
        term2string(Y,S2), list_concat(['(',S1, ' V ',S2,')'],S).
term2string(y(X,Y),S)  :- term2string(X,S1),
        term2string(Y,S2), list_concat(['(',S1, ' & ',S2,')'],S).
term2string(k(I,X),S)  :- term2string(X,S0),
        list_concat(['K',I,S0],S).
term2string(p(I,X),S)  :- term2string(X,S0),
        list_concat(['P',I,S0],S).
term2string(c(X),S)    :- term2string(X,S0),
        string_concat('C',S0,S).
term2string(u(X),S)    :- term2string(X,S0),
        string_concat('U',S0,S).
term2string(a(X),S)    :- term2string(X,S0),
        string_concat('C[H]',S0,S).
```

A.4 The Proof Module

The mod_proof module implements the rules of the system.

```
:- module(proof,[id_rule/1, rep_rule/2, o_rule/2, y_rule/3, u_rule/2,
        c_rule/3, a_rule/3,foc_rule/4,k_rule/2,locally_reduced/1]).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% write_annot(L,OO,X) :- writes the annotation L (one context for      %%
%%               line) in stream OO if X=1; otherwise does nothing.      %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

write_annot(_,_,0)      :- !.
write_annot([],OO,1)    :- !, write(OO,'H = []'), nl(OO).
write_annot([H|T],OO,1) :- list_concat(['H = [[',H,']'],St),
        write(OO,St), write_annot(T,OO).
write_annot([],OO)      :- write(OO,']'), nl(OO).
write_annot([H|T],OO)   :- nl(OO), list_concat(['     [',H,']'],S),
        write(OO,S), write_annot(T,OO).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Rules of the proof-system. %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% id_rule(L) :- true if L is an instance of ID; fails otherwise.       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

id_rule(L) :- member(plit(X),L), member(nlit(X),L), !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% rep_rule(L,A) :- true if some context of list A is included in L     %%
%%               and contains a c-formula; fails otherwise.              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rep_rule(L,A) :- get_uset(L,C), !, includes(C,A).

get_uset(L,C) :- drop(L,a(_),C).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% o_rule(L1,L2) :- true if L1 is the conclusion of rule OR applied to  %%
%%              L2; fails otherwise.                                     %%
%% y_rule(L1,L2;L3) :- true if L1 is the conclusion of rule AND applied %%
%%              to L2 and L3, fails otherwise.                           %%
%% u_rule(L1,L2) :- true if L1 is the conclusion of rule U applied to   %%
%%              L2; fails otherwise.                                     %%
%% c_rule(L1,L2,L3) :- true if L1 is the conclusion of rule C applied   %%
%%              to L2 and L3; fails otherwise.                           %%
%% a_rule(L1,L2,L3) :- true if L1 is the conclusion of rule A applied   %%
%%              to L2 and L3; fails otherwise.                           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

o_rule(L1,L2)               :- o_rule1(L1,L2a), no_dup(L2a,L2b),
        sort_list(L2b,L2).
o_rule1([],_)               :- !, fail.
o_rule1([o(X,Y)|T],[X,Y|T]) :- !.
o_rule1([H|T1],[H|T2])      :- !, o_rule1(T1,T2).

y_rule(L1,L2,L3)                :- y_rule1(L1,L2a,L3a),
        no_dup(L2a,L2b), no_dup(L3a,L3b), sort_list(L2b,L2),
        sort_list(L3b,L3).
y_rule1([],_,_)                 :- !, fail.
y_rule1([y(X,Y)|T],[X|T],[Y|T]) :- !.
y_rule1([H|T1],[H|T2],[H|T3])   :- !, y_rule(T1,T2,T3).

u_rule(L1,L2)               :- u_rule1(L1,L2a), no_dup(L2a,L2b),
sort_list(L2b,L2).
u_rule1([],_)               :- !, fail.
u_rule1([u(X)|T],[E1,E2|T]) :- !, u_unfold(X,E1,E2).
u_rule1([H|T1],[H|T2])      :- !, u_rule1(T1,T2).

c_rule(L1,L2,L3)                  :- c_rule1(L1,L2a,L3a),
        no_dup(L2a,L2b), no_dup(L3a,L3b), sort_list(L2b,L2),
        sort_list(L3b,L3).
c_rule1([],_,_)                   :- !, fail.
c_rule1([c(X)|T],[E1|T],[E2|T])   :- !, c_unfold(X,E1,E2).
c_rule1([H|T1],[H|T2],[H|T3])     :- !, c_rule1(T1,T2,T3).
```

```
a_rule(L1,L2,L3)                  :- a_rule1(L1,L2a,L3a),
        no_dup(L2a,L2b), no_dup(L3a,L3b), sort_list(L2b,L2),
        sort_list(L3b,L3).
a_rule1([],_,_)                   :- !, fail.
a_rule1([a(X)|T],[E1|T],[E2|T])   :- !, a_unfold(X,E1,E2).
a_rule1([H|T1],[H|T2],[H|T3])     :- !, a_rule1(T1,T2,T3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% foc_rule(L1,L2,P1,P2) :- L2 is L1 with a focused formula. This is    %%
%%              the first formula in the priority list P1, which        %%
%%              becomes P2 when the chosen element is pushed to the     %%
%%              end of the list; fails if there are no focusable        %%
%%              formulae.                                               %%
%% foc_rule(L1,L2,X) :- L2 is L1 with formula X focused; if X is not    %%
%%              in L1, the predicate fails.                             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

foc_rule(_,_,[],_)                :- !, fail.
foc_rule(L1,L2,[H|T],L)           :- foc_rule(L1,L1a,H), !,
        append(T,[H],L), sort_list(L1a,L2).
foc_rule(L1,L2,[H|T1],[H|T2])     :- !, foc_rule(L1,L2,T1,T2).
foc_rule([],_,_)                  :- !, fail.
foc_rule([c(X)|T],[a(X)|T],c(X)) :- !.
foc_rule([H|T1],[H|T2],F)         :- !, foc_rule(T1,T2,F).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% k_rule(L1,L2,N) :- L2 is a list with terms ap(N,P) where N is an     %%
%%              agent and P is a premise for this agent by rule K with  %%
%%              conclusion L1, and N is the total number of premises.   %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k_rule(L,Lx) :- locally_reduced(L), !, get_kp(L,L1,L2),
        get_premises(L1,L2,Lxa), sort_list2(Lxa,Lx).

sort_list2([],[]).
sort_list2([ap(A,L1)|T1],[ap(A,L2)|T2]) :- sort_list(L1,L2),
        sort_list2(T1,T2).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% locally_reduced(L) :- succeeds if L is locally reduced and has at    %%
%%              least one k-formula; fails otherwise.                   %%
%% get_kp(L,L1,L2) :- L1 contains all k-formulae of L and L2 contains   %%
%%              all p-formulae of L. The clause fails if L is not in    %%
%%   KNF.                                                               %%
%% get_premises(L1,L2,L3,) :- L1 is a list of k-formulae and L2 is a    %%
%%              list of p-formulae. L3 is a list with one premise of a  %%
%%              k-formula of L1 and all all premises of the p-formulae  %%
%%              of L2 corresponding to  the same agent.                 %%
%% get_p(N,L1,L2) :- N is an agent, L1 is a list of p-formulae and L2   %%
%%              is the list of p-formulae corresponding to agent N.     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

locally_reduced(L) :- member(k(_,_),L), quasi_locally_reduced(L).
quasi_locally_reduced([]).
quasi_locally_reduced([H|T]) :- quasi_lr(H), quasi_locally_reduced(T).

quasi_lr(p(_,_)).
quasi_lr(k(_,_)).
quasi_lr(plit(_)).
quasi_lr(nlit(_)).

get_kp([],[],[]).
get_kp([k(N,X)|T],[k(N,X)|T1],L2) :- get_kp(T,T1,L2).
get_kp([p(N,X)|T],L1,[p(N,X)|T2]) :- get_kp(T,L1,T2).
get_kp([plit(_)|T],L1,L2)         :- get_kp(T,L1,L2).
get_kp([nlit(_)|T],L1,L2)         :- get_kp(T,L1,L2).

get_premises([],_,[]).
get_premises([k(N,X)|T1],L,[ap(N,[X|T2])|T3]) :- get_p(N,L,T2),
        get_premises(T1,L,T3).

get_p(_,[],[]).
get_p(N,[p(N,X)|T1],[X|T2]) :- !, get_p(N,T1,T2).
get_p(N,[_|T],L)            :- !, get_p(N,T,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% legend(Kx,St) :- writes the codes Kx of the premises of rule K       %%
%%              in St.                                                  %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
legend([K],St)       :- !,
        list_concat(['K rule yields node <',K,'>'],St).
legend([K|T],St)     :- !,
        list_concat(['K rule yields nodes <',K,'>'],St0),
        legend(T,St0,St).
legend([K],St0,St)   :- !, list_concat([St0,' and <',K,'>'],St).
legend([K|T],St0,St) :- !, list_concat([St0,', <',K,'>'],St1),
        legend(T,St1,St).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% u_unfold(X,E1,E2) :- a formula X is "unfolded" into two              %%
%%              disjunctions for all agents; this corresponds to the    %%
%%              active formulae in the premise of rule U.               %%
%% c_unfold(X,E1,E2) :- a formula X is "unfolded" into two              %%
%%              conjunctions for all agents; this corresponds to the    %%
%%              active formulae in the premise of rule C.               %%
%% a_unfold(X,E1,E2) :- same as c_unfold/3 for annotated formulae.      %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

u_unfold(X,E1,E2)                                 :- agents(L),
        u_unfold(X,E1,E2,L).
u_unfold(X,p(N,X),p(N,u(X)),[N])                  :- !.
u_unfold(X,o(p(N,X),Z1),o(p(N,u(X)),Z2),[N|T])    :- !,
        u_unfold(X,Z1,Z2,T).

c_unfold(X,E1,E2)                                 :- agents(L),
        c_unfold(X,E1,E2,L).
c_unfold(X,k(N,X),k(N,c(X)),[N])                  :- !.
c_unfold(X,y(k(N,X),Z1),y(k(N,c(X)),Z2),[N|T])    :- !,
        c_unfold(X,Z1,Z2,T).

a_unfold(X,E1,E2)                                 :- agents(L),
        a_unfold(X,E1,E2,L).
a_unfold(X,k(N,X),k(N,a(X)),[N])                  :- !.
a_unfold(X,y(k(N,X),Z1),y(k(N,a(X)),Z2),[N|T])    :- !,
        a_unfold(X,Z1,Z2,T).
```

A.5 The Services Module

The mod_services module implements the code for general-purpose predicates, such as concatenation or sorting of lists.

```
:-module(services,[member/2,no_dup/2,append/3,list_concat/2,includes/2,
        drop/3,sort_list/2,no_atoms/2,list_length/2,list_max/2,
        get_agents/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% member(X,L) :- Element X is a member of list L.                          %%
%% no_dup(L1,L2) :- L2 contains the elements of L1 without                  %%
%%              duplicates.                                                 %%
%% append(L1,L2,L3) :- L3 is the concatenation of lists L1 and L2.          %%
%% list_concat(L,X) :- string X is the concatenation of the strings of      %%
%%              list L.                                                     %%
%% sort_list(L1,L2) :- L2 has the same elements as L2 sorted by the         %%
%%              standard order @< of terms:                                 %%
%%              variables@<numbers@<atoms@<strings@<structures@<lists       %%
%% list_length(L,N) :- N is the number of elements of list L.               %%
%% list_max(L,Mx) :- Mx is the height of the highest code in the list       %%
%%              of codes L.                                                 %%
%% get_agents(L,S) :- L is a list of agents and S is a string               %%
%%              containing the agents separated by commas.                  %%
%% drop(L1,X,L2) :- List L2 is list L1 without the element X. Fails         %%
%%              if X is not a member of L1.                                 %%
%% includes(X,L) :- X is a superset of some element of list L. Fails        %%
%%              if this is not the case.                                    %%
%% inc(L1,L2) :- List L1 is a superset of list L2. Both lists are           %%
%%              ordered according to the standard order @< of terms         %%
%%              (see abbove.)                                               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

includes(X,[H])   :- !, inc(X,H).
includes(L,[H|_]) :- inc(L,H), !.
includes(L,[_|T]) :- !, inc(L,T).

drop([H|T],H,T)       :- !.
drop([H|T1],X,[H|T2]) :- !, drop(T1,X,T2).
```

```
inc(_,[])          :- !.
inc([H|T1],[H|T2]) :- !, inc(T1,T2).
inc([_|T1],L2)     :- !, inc(T1,L2).

member(H,[H|_]).
member(X,[_|T]) :- member(X,T).

no_dup([],[]).
no_dup([H|T],L)       :- member(H,T), no_dup(T,L).
no_dup([H|T1],[H|T2]) :- no_dup(T1,T2).

append([],L,L).
append([H|T1],L,[H|T2]) :- append(T1,L,T2).

list_concat([],'').
list_concat([H|T],X) :- list_concat(T,X1), string_concat(H,X1,X).

sort_list(L1,L2) :- swap(L1,L1a), !, sort_list(L1a,L2).
sort_list(L,L).

swap(L1,L2) :- append(Pre,[X1,X2|T],L1), X1@>X2, !,
        append(Pre,[X2,X1|T],L2).

no_atoms([],[]).
no_atoms([plit(_)|T1],L2) :- !, no_atoms(T1,L2).
no_atoms([nlit(_)|T1],L2) :- !, no_atoms(T1,L2).
no_atoms([H|T1],[H|T2])   :- !, no_atoms(T1,T2).

list_length(L,N) :- list_length(L,N,0).
list_length([],N,N).
list_length([_|T],N,Ac) :- plus(Ac,1,Ac1), list_length(T,N,Ac1).

list_max(L,N) :- list_max(L,N,0).
list_max([],Mx,Mx).
list_max([H|T],Mx,Ac) :- >(H,Ac), !, list_max(T,Mx,H).
list_max([_|T],Mx,Ac) :- !, list_max(T,Mx,Ac).

get_agents([H|T],S) :- get_agents(T,S,H).
get_agents([],S,S).
get_agents([H|T],S,S1) :- list_concat([S1,', ',H],S2),
        get_agents(T,S,S2).
```

A.6 The Reports Module

The mod_reports module implements the code for the report about the preproof (or proof), the display of the tree, and statistics (number of nodes, height of the tree, time elapsed.) It contains also the code for the extraction of a proof or of a countermodel.

```
:- module(reports,[get_derivation/1,get_preproof/1,get_model/1,sum_up/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_model(OO) :- this predicate constructs a (counter-) model; OO    %%
%%               is the output stream.                                   %%
%% get_model(S,R,V,OO) :- S is a list of states of the form 's0',        %%
%%               's1', and so on; R is a list of lists of binary         %%
%%               relations on the set of  states of the form             %%
%%               'r(s0,s1)'; there is a list of relations for each       %%
%%               agent; V is a list of terms  of the form 'v(s0,L)'      %%
%%               where L is a list of propositions; OO is the output     %%
%%               stream.                                                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_model(OO) :- get_premodel(S,R), retractall(mx(_)), get_sv(S,Ms,Mv),
        prepare(Mr0), get_r(S,R,Mr0,Mr),
        write(OO, '** countermodel: M=(S,R,V)'), nl(OO), nl(OO),
        write_states(Ms,OO), write_relations(Mr,OO),
        write_valuations(Mv,OO).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_premodel(S,R) :- this is the first step to get the              %%
%%              countermodel; S is a list of terms of the form         %%
%%              'st(N,P,K)' where N is the number of a state, P is a   %%
%%              list of propositions that hold in that state and K is  %%
%%              a list of nodes grouped in the state; the current      %%
%%              state is stored in the dynamic predicate 'mx(N)'; R    %%
%%              is a list of terms of the form 'br(A,K,K0)' where A is %%
%%              an agent and (K,K0) is a relation for this agent.      %%
%% get_premodel(S,T,R,K) :- S and R as before; T is at the beginning   %%
%%              an term of the form 'st(N,K)' with N and K as above    %%
%%              and K empty. K is the code of the node being analysed. %%
%% deriv(L,S1,S2,K,Rule,X,R,Succ) :- P is the list of negative         %%
%%              literals occurring in the sequent; S1, S2, K, and X as %%
%%              before; 'Rule' is the rule applied to the sequent and  %%
%%              the list 'Succ' has the codes of the successors. This  %%
%%              is a dynamic predicate of the main program.            %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_premodel(S,R) :- assert(mx(0)), get_premodel(S,st(0,[]),R,0).

get_premodel([st(N,P,[K|Kx])],st(N,Kx),[],K) :-
        deriv(P,_,_,K,'i',_,1,_,_).

get_premodel([st(N,P,[K|Kx])],st(N,Kx),R,K) :-
        deriv(P,_,_,K,'o',_,1,[K0],_), get_cyclic_relations(R,K0,K).

get_premodel(S,st(N,Kx),R,K) :- deriv(_,_,_,K,Rul,_,1,[K0],_),
        rule1(Rul), get_premodel(S,st(N,[K|Kx]),R,K0).
get_premodel(S,st(N,Kx),R,K) :- deriv(_,_,_,K,Rul,_,1,[K0,K1],_),
        rule2(Rul), choose(K0,K1,Knxt),
        get_premodel(S,st(N,[K|Kx]),R,Knxt).

get_premodel([st(N,P,[K|Kx])|T],st(N,Kx),R,K) :-
        deriv(P,_,_,K,'BOX',_,1,Lx,_),
        get_relations(R2,Lx,K), get_nxt_states(T,R1,Lx), append(R1,R2,R).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_nxt_states(S,R,Nxt) :- get the following states when a 'BOX'     %%
%%               rule occurs; S and R as in the predicate               %%
%%               'get_premodel/2', and Nxt is the list of elements of   %%
%%               the form 'ap(A,K)' of the predicate deriv/8 (recall A  %%
%%               is an agent and K is a code.)                          %%
%% get_relations(R,Nx,K) :- creates a list of elements of the form      %%
%%               'br(A,K0,K1)' where A is an agent and K0 and K1 are    %%
%%               codes of nodes; Nx is a list of elements of the form   %%
%%               'ap(A,K)', as before and K is a code.                  %%
%% get_cyclic_relations(R,K0,K) :- similar to the previous one, but     %%
%%               for cyclic relations; K0 is the code of the cyclic     %%
%%               node, K is code of the node and R as before.           %%
%% choose(K1,K2,K) :- all parameters are codes of nodes; K is either    %%
%%               K1 or K2; it must be unsuccessful.                     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_nxt_states([],[],[]).
get_nxt_states(S1,R,[ap(_,K)|T2]) :- retract(mx(N)), plus(N,1,N1),
        assert(mx(N1)),
        get_premodel(S,st(N1,[]),R1,K), get_nxt_states(T1,R2,T2),
        append(S,T1,S1), append(R1,R2,R).

get_relations([],[],_).
get_relations([br(N,K,K0)|T1],[ap(N,K0)|T2],K) :- get_relations(T1,T2,K).

get_cyclic_relations(R,K0,K) :- deriv(_,_,_,K0,_,_,_,Lx,_),
        get_relations(R,Lx,K).

choose(K,_,K) :- deriv(_,_,_,K,_,_,1,_,_), !.
choose(_,K,K).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% prepare(Mr) :- produces a list of N empty lists, one for each        %%
%%               agent.                                                 %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

prepare(Mr0) :- agents(L), prepare(Mr0,L).
prepare([],[]).
prepare([[]|T1],[_|T2]) :- prepare(T1,T2).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_sv(L1,L2,L3) :- L1 is a list of elements of the form            %%
%%             'st(N,P,K)', as in get_premodel/2; L2 is a list of      %%
%%             elements of the form 's0', 's1', and so on; L3 is a     %%
%%             list of the form 'v(SN,P)' where SN is an element of    %%
%%             L2 and P is a list of propositions; SN is a state and   %%
%%             the elements of P are the propositions that hold in     %%
%%             SN.                                                      %%
%% get_r(L1,L2,L3,L4) :- L1 is as above; L2 is a list of elements of    %%
%%             the form 'br(A,K0,K1)' where A is an agent and K0 and   %%
%%             K1 are codes of the nodes;  L3 is an accumulator and    %%
%%             L4 is the the resulting list of lists of elements of    %%
%%             the form 'r(S1,S2)', where S1 and S2 are states.        %%
%% get_element(N,L,X) :- X is the N-th element of list L.               %%
%% get_state(K1,S1,S) :- as above for individual K1 and S1.             %%
%% add_to_list(L1,X,N,L2) :- L1 is a list of lists; X is added to the   %%
%%             N-th list and L2 is the result.                          %%
%% list_concat(L,S) :- L is a list of strings and S is the              %%
%%             concatenation of them all.                               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_sv([],[],[])             :- !.
get_sv([st(N,P,_)|T1],[SN|T2],[v(SN,P)|T3]) :- !,
        string_concat('s',N,SN), get_sv(T1,T2,T3).

get_r(_,[],R,R).
get_r(S,[br(N,K1,K2)|T],Mr0,Mr) :- get_state(K1,S1,S),
        get_state(K2,S2,S), atom_number(N,Nx),
        add_to_list(Mr0,r(S1,S2),Nx,Mr0a), get_r(S,T,Mr0a,Mr).

get_state(K,SN,[st(N,_,Kx)|_]) :- member(K,Kx), !,
        string_concat('s',N,SN).
get_state(K,N,[_|T]) :- !, get_state(K,N,T).

add_to_list([H|T],X,1,[[X|H]|T]) :- !.
add_to_list([H|T1],X,N,[H|T2])  :- !, plus(M,1,N),
        add_to_list(T1,X,M,T2).
```

```
list_concat(L,S) :- list_concat(L,S,'').

list_concat([],S,S).
list_concat([H|T],S,S0) :- string_concat(S0,H,S1),
        list_concat(T,S,S1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% write_states(Ms,OO) :- Ms is a list of states of the form 's0',      %%
%%              's1', and so on and OO is an output stream.            %%
%% write_relations(Mr,OO) :- Mr is a list of relations of the form      %%
%%              'r(s0,s1)' where 's0', 's1' as above, and OO is an     %%
%%              output stream.                                         %%
%% write_valuations(Mv,OO) :- Mv is a list of valuiations of the form   %%
%%              'v(s0,P)' where 's0' as above and P is the list of the %%
%%              propositions that hold in 's0'.                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

write_states(Ms,OO) :- sort(Ms,Ms0), get_string_s(Ms0,St0,''),
        string_concat('S = {',St0,St),
        write(OO,St), nl(OO), nl(OO).

write_relations(Mr,OO) :- agents(L), get_string_r(L,R,'R = {'),
        write(OO,R), nl(OO), write_rel1(Mr,L,OO), nl(OO).

write_valuations(Mv,OO) :- write(OO,'V = {'), write_valuations(Mv,OO,0).
write_valuations([],OO,_) :- write(OO,'}'),nl(OO),nl(OO).
write_valuations([v(S,P)|T],OO,0) :- list_concat(['(',S,',['],S1),
        get_string_v(P,S2,''), list_concat([S1,S2,'])'],St),
        write(OO,St), write_valuations(T,OO,1).
write_valuations([v(S,P)|T],OO,1) :- write(OO,','), nl(OO),
        list_concat(['     (',S,',['],S1),
        get_string_v(P,S2,''), list_concat([S1,S2,'])'],St),
        write(OO,St), write_valuations(T,OO,1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_preproof(OO) :- this predicate constructs a preproof; OO is the  %%
%%              output stream.                                         %%
%% get-preproof(K,OO) :- K is the code of the node and OO is the        %%
%%              output stream.                                         %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
get_preproof(OO) :- get_preproof(0,OO).

get_preproof(K,OO) :- deriv(_,S1,S2,K,'ID',X,_,_,_),
        list_concat(['Node <',K,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,X),
        write(OO,'instance of ID -- Node successful'),
        nl(OO), nl(OO).

get_preproof(K,OO) :- deriv(_,S1,S2,K,'REP',1,_,_,_),
        list_concat(['Node <',K,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,1),
        write(OO,'instance of REP -- Node successful'),
        nl(OO), nl(OO).

get_preproof(K,OO) :- deriv(_,S1,_,K,'i',0,_,_,_),
        list_concat(['Node <',K,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO),
        write(OO,'irreducible node -- Node unsuccessful'),
        nl(OO), nl(OO).

get_preproof(K,OO) :- deriv(_,S1,S2,K,'o',X,_,[KO],_),
        list_concat(['Node <',K,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,X),
        list_concat(['cyclic node: twin node is <',KO,
        '> -- Node unsuccessful'],St2),
        write(OO,St2), nl(OO), nl(OO).

get_preproof(K,OO) :- deriv(_,S1,S2,K,Rule,X,_,[Nxt],_),
        rule1(Rule), list_concat(['Node <',K,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,X),
        list_concat([Rule,' rule yields node <',Nxt,'>'],St2),
        write(OO,St2), nl(OO), nl(OO),
        get_preproof(Nxt,OO).

get_preproof(K,OO) :- deriv(_,S1,S2,K,Rule,X,_,[Nxt1,Nxt2],_),
        rule2(Rule), list_concat(['Node <',K,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,X),
        list_concat([Rule,' rule yields nodes <',
                    Nxt1,'> and <',Nxt2,'>'],St2),
        write(OO,St2), nl(OO), nl(OO),
        get_preproof(Nxt1,OO), get_preproof(Nxt2,OO).
```

```
get_preproof(K,OO) :- deriv(_,S1,S2,K,'BOX',X,_,L,_),
        get_line(L,St2),
        list_concat(['Node <',K,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,X),
        write(OO,St2), nl(OO), nl(OO),
        get_preproofs(L,OO).

get_preproofs([],_).
get_preproofs([ap(_,K)|T],OO) :- get_preproof(K,OO),
        get_preproofs(T,OO).

get_line([ap(_,K)],S) :- !,
        list_concat(['BOX rule yields node <',K,'>'],S).
get_line([ap(_,K)|T],S):- !,
        list_concat(['BOX rule yields nodes <',K,'>'],S1),
        get_line(T,S1,S).

get_line([],S,S).
get_line([ap(_,K)|T],S1,S) :- list_concat([S1,', <',K,'>'],S2),
        get_line(T,S2,S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_derivation(OO) :- this predicate construct a derivation; OO is   %%
%%              the output stream.                                      %%
%% get_derivation(K,C,OO) :- K is the code of the node, C is the        %%
%%              binary node(recall that there are multiple nodes        %%
%%              in 'deriv'), and OO is the output stream.               %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_derivation(OO) :- assert(dcode(0)), get_derivation(0,0,OO),
        retractall(dcode(_)).

get_derivation(K,C,OO) :- deriv(_,S1,S2,K,'ID',X,_,_,_),
        list_concat(['Node <',C,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,X),
        write(OO,'instance of ID -- Node successful'),
        nl(OO), nl(OO).
```

```
get_derivation(K,C,OO) :- deriv(_,S1,S2,K,'REP',1,_,_,_),
        list_concat(['Node <',C,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,1),
        write(OO,'instance of REP -- Node successful'),
        nl(OO), nl(OO).

get_derivation(K,C,OO) :- deriv(_,S1,S2,K,Rule,X,0,[Nxt],_),
        rule1(Rule), get_dcode(C0),
        list_concat(['Node <',C,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO), write_annot(S2,OO,X),
        list_concat([Rule,' rule yields node <',C0,'>'],St2),
        write(OO,St2), nl(OO), nl(OO),
        get_derivation(Nxt,C0,OO).

get_derivation(K,C,OO) :- deriv(_,S1,S2,K,Rule,X,0,[Nxt1,Nxt2],_),
        rule2(Rule), get_dcode(C0), get_dcode(C1),
        list_concat(['Node <',C,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO),
        write_annot(S2,OO,X),
        list_concat([Rule,' rule yields nodes <',
                    C0,'> and <',C1,'>'],St2),
        write(OO,St2), nl(OO), nl(OO),
        get_derivation(Nxt1,C0,OO), get_derivation(Nxt2,C1,OO).

get_derivation(K,C,OO) :- deriv(_,S1,S2,K,'BOX',X,0,L,_),
        get_nxt(L,Nxt), get_dcode(C0),
        list_concat(['Node <',C,'> = [',S1,']'],St1),
        write(OO,St1), nl(OO),
        write_annot(S2,OO,X),
        list_concat(['BOX rule yields node <',C0,'>'],St2),
        write(OO,St2), nl(OO), nl(OO),
        get_derivation(Nxt,C0,OO).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_dcode(X) :- gets the next available code for the derivation.     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_dcode(C) :- dcode(X), plus(X,1,C), retract(dcode(X)),
        assert(dcode(C)).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% write_annot(L,OO,X) :- writes the annotation L (one context for      %%
%%               line) in stream OO if X=1; otherwise does nothing.     %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

write_annot(_,_,0)      :- !.
write_annot([],OO,1)    :- !, write(OO,'H = []'), nl(OO).
write_annot([H1,H2|T],OO,1) :- !, list_concat(['H = [[',H1,'],'],St),
        write(OO,St), write_annot([H2|T],OO).
write_annot([H|T],OO,1) :- !, list_concat(['H = [[',H,']'],St),
        write(OO,St), write_annot(T,OO).
write_annot([],OO)      :- write(OO,']'), nl(OO).
write_annot([H|T],OO)   :- nl(OO), list_concat(['     [',H,']'],S),
        write(OO,S), write_annot(T,OO).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% rule1(X) :- X is a rule with one premise other than K.               %%
%% rule2(X) :- X is a rule with two premises other than K.              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rule1('OR').
rule1('U').
rule1('FOC').
rule2('AND').
rule2('C').
rule2('C[H]').

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_nxt(L,N) :- L is a list of premises of a BOX rule and N is a     %%
%%               successful one.                                        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_nxt([ap(_,H)|_],H) :- deriv(_,_,_,H,_,_,0,_,_).
get_nxt([_|T],N) :- get_nxt(T,N).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% get_string_s(L,S,S0) :- prepares a string S with the elements of     %%
%%              list L; S0 is an  accumulator.                          %%
%% write_rel1(L1,L2,OO) :- writes the relations in list L1 (of the      %%
%%              form 'r(s0,s1)') for the agents in list L2 on output    %%
%%              stream OO.                                              %%
%% write_rel2(L,S,St0) :- prepares the string S to be written for the   %%
%%              relations in list L, of the same form as above; St0 is  %%
%%              an accumulator.                                         %%
%% get_string_r(L,S,S0) :- similar to get_string/3, bt for lists of     %%
%%              relations.                                              %%
%% get_string_v(L,S,S0) :- similar to get_string/3 for lists of         %%
%%              valuations.                                             %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_string_s([],St,St0)    :- string_concat(St0,'}',St).
get_string_s([H|T],St,'')  :- !, get_string_s(T,St,H).
get_string_s([H|T],St,St0) :- !, list_concat([St0,',',H],St1),
        get_string_s(T,St,St1).

write_rel1(_,[],_).
write_rel1([L|T1],[A|T2],OO) :- list_concat(['  R',A,'= {'],St0),
        write_rel2(L,S,St0), write(OO,S), nl(OO), write_rel1(T1,T2,OO).

write_rel2([],S,St0) :- !, string_concat(St0,'}',S).
write_rel2([r(S0,S1)],S,St0) :- !,
        list_concat([St0,'(',S0,',',S1,')}'],S).
write_rel2([r(S0,S1)|T],S,St0) :-
        list_concat([St0,'(',S0,',',S1,'),'],St1),
        write_rel2(T,S,St1).

get_string_r([],R,S) :- string_concat(S,'}',R).
get_string_r([A],R,S) :- !, list_concat([S,'R',A,'}'],R).
get_string_r([A|T],R,S) :- !, list_concat([S,'R',A,','],S0),
        get_string_r(T,R,S0).

get_string_v([],S,S).
get_string_v([P|T],S,'') :- !, get_string_v(T,S,P).
get_string_v([P|T],S,S0) :- list_concat([S0,', ',P],S1),
        get_string_v(T,S,S1).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% sum_up(OO,T0) :- this predicate writes the total number of nodes,    %%
%%               the maximum height of the tree and the (roughly) time   %%
%%               elapsed regardless of the validity of the sequent; OO   %%
%%               is the  output stream and T0 is the initial time.       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sum_up(OO,T0) :- findall(C,deriv(_,_,_,C,_,_,_,_,_,_),L1),
        list_length(L1,N), findall(H,deriv(_,_,_,_,_,_,_,_,_,H),L2),
        list_max(L2,H), get_time(T1), T is round((T1 - T0) * 1e+06),
        list_concat(['** Total number of nodes      = ',N],S1),
        list_concat(['** Maximum height of the tree = ',H],S2),
        list_concat(['** Time elapsed               = ',
                     T,'e-06 sec'],S3),
        write(OO,S1), nl(OO), write(OO,S2), nl(OO), write(OO,S3),
        nl(OO), nl(OO).
```

Appendix B

The Construction of a Coutermodel

B.1 Construction of a Countermodel in S_{CK}

The procedure to construct the countermodel when the sequent is not provable is the most complicated of the implementation. The decision procedure and the construction of a proof in S_{CK} starting from one in S'_{CK} are comparatively straightforward. The problem lies in the conversion between different trees that are not isomorphic. This process was explained in Chapter 3. The extraction of the countermodel is in module mod_reports as is the procedure for the construction of a proof. The construction of a countermodel is a more complicated process than the construction of a proof. We give here a general explanation highlighting the more relevant parts without incurring in many technical details. The complete code is in Appendix A.

The countermodel is constructed following Definition 3.49 of Chapter 3. The output of the program is in the case of a sequent that is not provable, the countermodel. An excerpt of the output of a failed proof for a simple case (the sequent $\{\Diamond\Box_1\neg p, \Box_1\neg p\}$) is shown in Figure B.1. In this case, some elements that are not essential have been omitted and replaced with ellipses (...); the complete listing of the output for this example may be

found in Chapter 5.

```
** Agents   [1, 2]
** Sequent  [UK1¬p,K1¬p]
Formula Nr.1: UK1¬p
OK
Formula Nr.2: K1¬p
OK
(...)
** countermodel: M=(S,R,V)
S = {s0,s1,s2}
R = {R1,R2}
  R1= {(s0,s1),(s1,s2),(s2,s2)}
  R2= {}
V = {(s0,[]),
     (s1,[p]),
     (s2,[p])}
** Total number of nodes        = 12
** Maximum height of the tree   = 12
** Time elapsed                 = 3273e-06 sec
** The sequent is not valid.
```

Fig. B.1: The output of a simple preproof.

The part of the proof that has been omitted in Figure B.1, denoted with ellipses (...), is the preproof. The basis of the process, as in the construction of a proof, is the table deriv/8, which stores the whole information of the preproof.

There are several problems that must be dealt with. For instance, a state in the countermodel comprises several nodes in the preproof; besides, not all nodes in the preproof are used, because of the pruning process when the tree τ_1 is constructed (see Definition 3.49 in Chapter 3.)

The first part is the construction of a premodel, which consists of two

lists:

- A list St of terms of the form st(N,P,Kx), where N is the code of the state in the countermodel, P is a list of the propositions that hold in the state according to Definition 3.49, and Kx is a list containing codes of the nodes of the preproof that are collapsed in the state N.

- A list Re of terms of the form br(A,K1,K2) where A is an agent, and K1 and K2 are codes of nodes in the preproof such that K2 is an A-premise of K1.

Example B.1 The meaning of the lists St and Re is shown in Figure B.2. We have used the same sequent as in the example of Figure B.1. To make the encoding of the nodes more readable, we write 0 for $\langle\rangle$, 1 for $\langle 0\rangle$ and so on.

♠

Observe that in the example above several nodes are grouped together in single states following Definition 3.49.

The principal predicates to construct this premodel are compose_state/4 and get_nxt_states/3. The simplified code of these predicates is shown in Figure B.3. As before, some auxiliary subgoals have been abbreviated and grouped as commentaries between symbols %.

The clause compose_state/4 has four parameters, (St, S, R, K). Their meaning is explained below:

- St is the list which will be unified with the list St of the premodel as explained above, i.e., it is a list of terms of the form st(N,P,Kx), where N is the code of the state in the model, P is a list of the propositions that hold in N and Kx is a list containing codes of all nodes collapsed in N.

- S is a term st(N,Kx) where N and Kx are the same as above. In this term the nodes collapsed in state N are iteratively collected in list Kx.

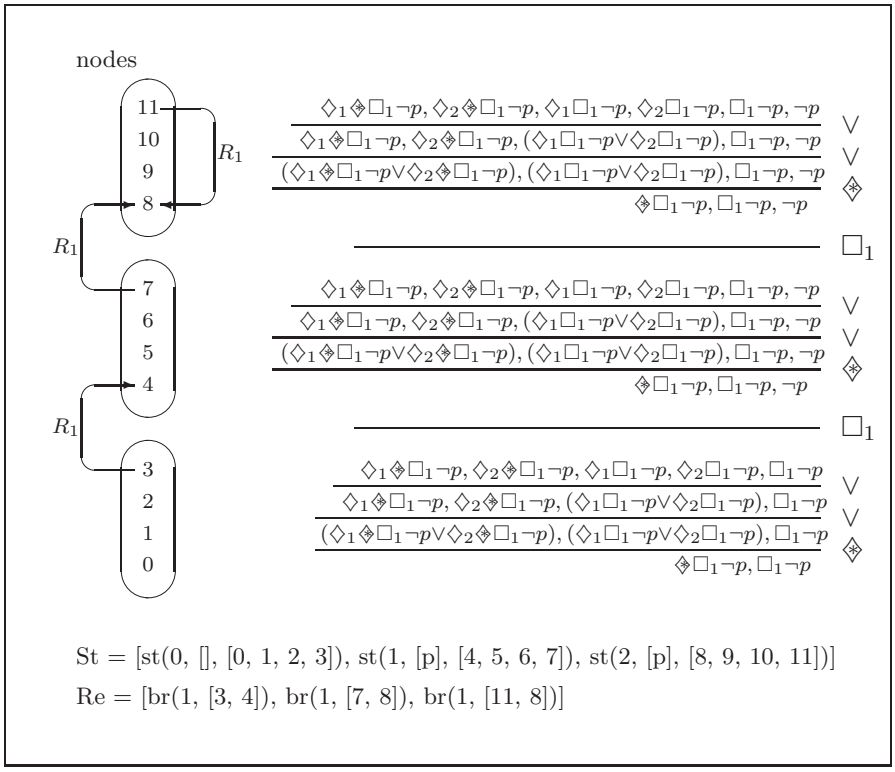


Fig. B.2: The first part of the construction of a countermodel given a failed derivation.

- R is a list of the same form as Re above, i.e., a list of terms br(A,K1,K2) where A is an agent, and K1 and K2 are codes of nodes in the preproof such that K2 is an A-premise of K1.
- The code K of the node being currently analysed.

The parameters of get_nxt_states/3 are:

- A list of the same form as St above.
- A list of the same form as Re above.

```
1.  get_premodel(S,R) :- assert(mx(0)), compose_state(S,st(0,[]),R,0).

2.  compose_state([st(N,P,[K|Kx])],st(N,Kx),[],K)      :-
        % The node is irreducible; P contains the atoms in N          %
3.  compose_state([st(N,P,[K|Kx])],st(N,Kx),R,K)       :-
        % The node is cyclic and R contains the relations from K      %
        % to the premises of the node that made K cyclic;             %
        % P contains the atoms in N                                   %
4.  compose_state(S,st(N,Kx),R,K)                      :-
        % A rule ∨′, ∧′, ⟐′, ⊡′, or ⊡′_H is applied to the node and    %
        % K0 is the code of an unsuccessful premise                   %
        compose_state(S,st(N,[K|Kx]),R,K0).
5.  compose_state([st(N,P,[K|Kx])|T],st(N,Kx),R,K)     :-
        % Rule □′ is applied to the node; P contains                  %
        % the propositions that hold in N and R2 contains             %
        % the relations from K to its premises                        %
        get_nxt_states(T,R1,Lx), append(R1,R2,R).

6.  get_nxt_states([],[],[]).
7.  get_nxt_states(S1,R,[ap(_,K)|T2])   :-
        % get the code of the new state (N1)                          %
        compose_state(S,st(N1,[]),R1,K), get_nxt_states(T1,R2,T2),
        append(S,T1,S1), append(R1,R2,R).
```

Fig. B.3: The first part of the contruction of a countermodel.

- A list of terms of the form ap(A,K) where A is an agent and K is a code of a node.

The construction of a countermodel proceeds as follows: first the clause get_premodel(S,st(0,[]),R,0) is called. Here S and R will be instantiated to the lists that constitute the premodel and the term st(0,[]) is an "empty" term which will collect the codes of all nodes in the state 0 of the countermodel. The fourth parameter is the code of the root of the preproof. Then there are several possibilities:

- If the branch is closed because an irreducible node has been encountered (clause 2), then the nodes collected in the second parameter together with the current node in the fourth one are put together in the list of the first parameter together with the list of all atomic propositions P that hold in state N. The branch is closed and no further recursive calls to compose_state/4 are done. The list R is empty for this branch, because there are no "next" states.

- If the branch is closed because a cyclic node has been encountered (clause 3), Then the nodes collected in the second parameter together with the current node in the fourth one are put together in the list of the first parameter together with the list of all atomic propositions P that hold in state N. The branch is closed and no further recursive calls to compose_state/4 are done. The list R contains the relations from the cyclic node to the premises of the node that caused the current node to be cyclic.

- If a rule other than $\Box'$ is applied to the node (clause 4), an unsuccessful premise K0 is chosen, the node is added to the collapsed nodes and a recursive call to compose_state4/ follows.

- If a rule $\Box'$ is applied to the node (clause 6), then the state must be closed and the next states processed. This is done as follows:

  - The head of the list unifies with the term st(N,P,[K|Kx]), where N is the code of the state being closed, P is the set of atomic propositions that hold in N, and [K|Kx] is the set of all nodes collapsed in N.
  - The predicate get_nxt_states(T,R1,Lx) is called. Here T and R1 will be unified with the remaining tail of the lists constituting the premodel. The get_nxt_states/3 predicate will call compose_state/4 predicate for each code of a premise node K in the list Lx, which consists of terms ap(A,K), where A is an agent.

A couple of observations are in order. In the first place, the predicate compose_state/4 takes only cyclic relations into account. The relations of

the underlying tree τ_1 (see Definition 3.49) have to be included after the call to the get_next_states/3 predicate. This is done in the last subgoal of clause 5. In the second place, there is a small technical problem with the codes of the states. It is clear that the first code is 0, but then several recursive calls to the compose_state/4 predicate behave as if they were independent processes [8] or independent threads [81]. Thus, a situation in which two calls to compose_state/4 occur with the same state code, i.e., with the same empty term st(N,[]) in the third parameter, must be avoided. This is done with a dynamic table mx/1, which has the current available code. The code abbreviated as "get the code of the new state" in clause 7 consists of taking the current value of mx/1 and updating it by incrementing the value in 1. This table plays the rôle of a global variable in usual procedural programming. In a multiprocess or multithreading implementation, this table should be treated as a mutual exclusion resource [8, 81].

The premodel contains the whole information of the model. Having the lists S and R, the extraction of the model is routine. The actual model consists of:

- A list Ms of states s0, s1, and so on.
- A list Mv of valuations, containing terms of the form v(S,P) where S is a state of the list Ms and P is a list of atomic propositions.
- A list Mr of lists of terms r(S1,S2), where S1 and S2 are states of the list Ms. The i-th list is the list of relations of agent i.

This is done with the predicates get_sv/3 and get_r/4, shown below. The predicate get_sv(S,Ms,Mv) instantiates Ms to the list of states and Mv to the list of valuations mentioned above. The predicate get_r(S,R,Mr0,Mr) where Mr0 is an auxiliary list of n empty lists, one for each of the n agents. instantiates Mr to the list of terms mentioned above. The predicates are shown in Figure B.4 below.

The predicate get_state(K,SN,S) instantiates the variable SN to the number corresponding to the state N where node K is collapsed and prefixes it with s. Now we show that the model constructed corresponds to Definition 3.49.

```
1.  get_sv([],[],[])                                              :- !.
2.  get_sv([st(N,P,_)|T1],[SN|T2],[v(SN,P)|T3])                   :- !,
      string_concat('s',N,SN), get_sv(T1,T2,T3).

3.  get_r(_,[],R,R).
4.  get_r(S,[br(N,K1,K2)|T],Mr0,Mr)                               :-
      get_state(K1,S1,S), get_state(K2,S2,S),
      atom_number(N,Nx), add_to_list(Mr0,r(S1,S2),Nx,Mr0a),
      get_r(S,T,Mr0a,Mr).

5.  get_state(K,SN,[st(N,_,Kx)|_]) :- member(K,Kx), !,
      string_concat('s',N,SN).
6.  get_state(K,N,[_|T]) :- !, get_state(K,N,T).

7.  add_to_list([H|T],X,1,[[X|H]|T]) :- !.
8.  add_to_list([H|T1],X,N,[H|T2])  :- !, plus(1,N,M),
      add_to_list(T1,X,M,T2).
```

Fig. B.4: The actual construction of the countermodel (Ms,Mr,Mv)**.**

In the following propositions we assume that there is a preproof whose information is stored in the table deriv/9, a corresponding premodel (S,R), and a corresponding model (Ms,Mr,Mv).

Proposition B.2 *Let* K0 *and* K1 *be the codes of two nodes in the preproof that have not been discarded in the construction of the premodel* (S,R)*. If* K0 *and* K1 *are connected and not separated for an instance of* $\Box'$*, then there is a term* (N,P,Kx) *in* S *such that both* K0 *and* K1 *are in* Kx*. Further,* K0 *and* K1 *do not appear in any other term of* S*.*

Proof. Both nodes are separated by instances of $\vee'$, $\wedge'$, foc$'$, $\diamondsuit'$, $\boxtimes'$, or $\boxtimes'_H$. When predicate compose_state/4 is called, clause 4 is repeated, since clause 2 and 3 close the branch (and therefore any connection), and clause 5 is only executed when there is an instance of rule $\Box'$. Thus, both codes will be stored in the third parameter of the predicate, the term st(N,Kx). When the branch is closed, either by clause 2, 3, or 5, the term is incorporated to

the first parameter under the code N of the state.

Observe besides that, once the predicate compose_state/4 is called, then a set of successor nodes is sent as third parameter to the get_next_states/3. In all cases, the nodes that are considered belong to a group which is separated from the already collapsed one by at least one instance of $\Box'$. The nodes that are below in the tree are not considered again. ■

Proposition B.3 *If the negative literal* nlit(p) *is in some node* K *which is collapsed in state* N*, then there is a term* v(N,P) *in* Mv *and* p *is in the list* P.

Proof. The first parameter of the table deriv/9 is a list of propositions which is constructed by the predicate get_prop/2, listed in Figure B.5 below.

```
1.  get_prop([],[]).
2.  get_prop([nlit(X)|T1],[X|T2])   :- !, get_prop(T1,T2).
3.  get_prop([_|T],P)               :- !, get_prop(T1,P).
```

Fig. B.5: The construction of the list of atomic propositions for the valuation of the countermodel.

The cuts here are red cuts. They are used to rule out undesired solutions. It is easy to see that for each negative literal nlit(X) in the sequent of the first parameter, X is in the list of the second parameter. This predicate is only called in the proof search process for sequents that are conclusions of the $\Box'$ rule. This is enough, since the rule $\Box'$ is the only one that can eliminate literals. Any other rule passes the literals from the conclusion on to the premises. If a negative literal nlit(X) is in a node K, and the node is the conclusion of a rule other than $\Box'$, it will be in all its premises. If it is the conclusion of $\Box'$, then clause 5 in the predicate compose_state/4 of Figure B.3 will be called and the state will be closed with a term st(N,P,Kx) where K is in Kx and X is in P. Besides, there is a term deriv(P,_,_,K,_,_,_,_,_).

When the predicate get_sv/3 is called, for each term st(N,P,_) in S, a term v(SN,P) is added to Mv, where SN is just N prefixed with s. ■

Proposition B.4 *Assume that the nodes* K0*,* K1*, and* K2 *have not been discarded in the construction of the premodel* (S,R)*. Assume further that* K0 *is collapsed in state* N0*,* K1 *is collapsed in state* N1 *and* K2 *is collapsed in state* N2*. Then:*

(i) *If* K1 *is an i-premise of* K0*, then there is a term* r(N0,N1) *in the i-th list of* Mr*.*

(ii) *If* twin (K1) = K0*, and* K2 *is an i-premise of* K0*, then there is a term* r(N0,N1) *in the i-th list of* Mr*.*

Proof. First we show for part (i) that there is a term br(i,K0,K1) in the list R of the premodel and for part (ii) that there is a term br(i,K1,K2) in R.

Part (i). If K1 is an i-premise of K0, then K0 is a conclusion of $\Box'$ and there is a term deriv(P,_,_,K0,'BOX',_,_,Lx,_), where Lx contains a term ap(i,K1). Then clause 5 in Figure B.3 calls predicate get_relations(R2,Lx,K0) with R2 uninstantiated. This predicate is listed in Figure B.6.

```
1.  get_relations([],[],_).
2.  get_relations([br(N,K,K0)|T1],[ap(N,K0)|T2],K) :- get_relations(T1,T2,K).
```

Fig. B.6: The obtention of the relations between two states of the countermodel.

It is easy to see that this predicate constructs a list of elements br(A,K0,K1) where A is an agent and K0 and K1 are the codes of two nodes. The idea is to specify that K1 is an A-premise of K0. Since there is a term ap(i,K1) in Lx, then there is a term br(i,K0,K1) in the list to which R2 will be instantiated.

Part (ii). If twin (K1) = K0, there is a term deriv(P,_,_,K1,'o',_,_,[K0],_). Besides, since we assume that K2 is an i-premise of K0, there is a term deriv(P,_,_,K0,'BOX',_,_,Lx,_), where the list Lx contains a term ap(i,K2). Then clause 3 in Figure B.3 succeeds and a subsequent call to the predicate

get_cyclic_relations(R,K0,K1) with R uninstantiated follows. This predicate just gets the aforementioned term deriv(P,_,_,K0,'BOX',_,_,Lx,_) and calls get_relations(R,Lx,K1) (see Figure B.6) with R uninstantiated. Thus, by part (i) of this proposition, there is a term br(i,K1,K2).

Now we show that there is a term r(N0,N1) in the i-th list of Mr. The list Mr is constructed by the predicate get_r/3 shown in Figure B.7 below. When a term br(i,K0,K1) is in R, the predicate add_to_list(Mr0,r(N0,N1),N,Mr0a) (clauses 5 and 6) is called. Here Mr0 is the list before the addition of the new element, r(N0,N1) is the term to be added to the list at position N, and Mr0a will be unified with the resulting list. It is easy to prove by a straightforward induction on N that the variable Mr0a unifies with the list that results from adding term r(N0,N1) to the list at position N of Mr0. The relevant clauses are shown below.

```
1.  get_r(_,[],R,R).
2.  get_r(S,[br(N,K1,K2)|T],Mr0,Mr)                    :-
        get_state(K1,S1,S), get_state(K2,S2,S),
        atom_number(N,Nx), add_to_list(Mr0,r(S1,S2),Nx,Mr0a),
        get_r(S,T,Mr0a,Mr).

3.  get_state(K,SN,[st(N,_,Kx)|_]) :- member(K,Kx), !,
        string_concat('s',N,SN).
4.  get_state(K,N,[_|T]) :- !, get_state(K,N,T).

5.  add_to_list([H|T],X,1,[[X|H]|T]) :- !.
6.  add_to_list([H|T1],X,N,[H|T2]) :- !, plus(1,N,M),
        add_to_list(T1,X,M,T2).
```

Fig. B.7: The construction of the list Mr**.**

■

Appendix C

Infix Operators

C.1 Implementation of Infix Operators

It would have been possible to define the terms in an infix form instead of the prefix form we used. As explained in Chapter 4, the main reason to choose the prefix form was that this form was to be used only in the inner manipulations of the formulæ and sequents and the user had another representation.

The main advantage would have been to have a unique representation of formulæ and thus avoid the problem of translating formulæ between the interface and the theorem prover. The main drawback was that Prolog imposes certain conditions in the syntax of operators, namely that they cannot begin with capital letters. Thus, there was the risk of having confusions betwen operators and atomic formulæ. This could have been overcome by using new symbols for the operators $\Box_i$, $\Diamond_i$, $\boxplus$ and $\diamondplus$. But then the objective of having a readable representation would have been rather compromised.

Still, the use of infix operators or prefix operators with a parsing process remains a matter of taste. Here we explain what has to be done to get infix operators.

The predicate to define a new operator is op/3, whose usage is the

following:

:- op(Precedence,Pattern,Symbol).

Here Precedence is an integer indicating the precedence; Pattern is either fx, xf, fy or yf for unary operators and xfx, xfy, xfy or yfy for binary operators. In these cases, x means a term of precedence strictly lower than the precedence of f and y means a term of precedence lower than or equal to that of f. Finally, Symbol is the name of the operator.

The op/3 predicate shold be executed before any use of the operator Symbol in the program. This definition may be done in a module and when the module is called, the operators are already defined. This is what we did in the next example.

Example C.1 We will show the general procedure by means of a simple example, a theorem prover for propositional logic. The program first reduces the formula to be proved to the negative normal form by eliminating the connectors $\equiv$ y $\Rightarrow$ and repeatedly applying the De Morgan laws afterwards. The connectors are: ~ (negation) \/ (a backslash and a slash; disjunction), /\ (a slash and a backslash; conjunction), => (implication) and <=> (equivalence.)

The module is defined thus:

```
:- module(mod_proof,
    [op(750,yfx,\/),
     op(760,yfx,/\),
     op(770,fx,~),
     op(780,yfx,=>),
     op(780,yfx,<=>),
     prove/2
    ]).
```

Fig. C.1: The definition of the module.

Notice that the predicate prove/2 and the operators are exported. The negation has precedence over all connectors, followed by conjunction, dis-

junction, implication and equivalence, the two latter with equal precedence. The pattern yfx means that connectors with the same precedence are left-associative.

The main program is shown next:

```
:- use_module(mod_proof).

attempt(F) :- prove(F,0), !, format('The formula is valid').
attempt(_) :- !, format('The formula is not valid').
```

Fig. C.2: The main module of the example.

For instance, two possible executions of the program are:

```
1 ?- attemt(p => (q => p))                                    .
The formula is valid
true.

2 ?- attemt(p => (q => a)).
The sequent is not valid
true.
```

Fig. C.3: Two runs of the program.

♠

Observe that the parentheses are necessary, since otherwise the formula

$$p_1 \Rightarrow p_2 \Rightarrow p_3$$

would be interpreted as

$$((p_1 \Rightarrow p_2) \Rightarrow p_3)$$

and not as

$$(p_1 \Rightarrow (p_2 \Rightarrow p_3))$$

as intended.

The complete code of the module, which is irrelevant to our purpose, is listed next.

```
:- module(mod_proof,
       [op(750,yfx,\/),
        op(760,yfx,/\),
    op(770,fx,~),
        op(780,yfx,=>),
        op(790,yfx,<=>),
        prove/2
       ]).

prove(F,R) :- convert_onto_nnf(F,F1), proof_search([F1],R).

proof_search(S,0) :- member(~X,S), member(X,S), !.
proof_search(S,R) :- or_rule(S,S1), proof_search(S1,R).
proof_search(S,R) :- and_rule(S,S1,S2), proof_search(S1,R1),
        proof_search(S2,R2), R is max(R1,R2).
proof_search(_,1) :- !.

convert_onto_nnf(F1,F2) :- elim_equiv(F1,F1a), elim_implic(F1a,F1b),
        get_nnf(F1b,F2).

elim_equiv(X,X) :- atom(X).
elim_equiv((~X),(~X1)) :- elim_equiv(X,X1).
elim_equiv((X \/ Y),(X1 \/ Y1)) :- elim_equiv(X,X1), elim_equiv(Y,Y1).
elim_equiv((X /\ Y),(X1 /\ Y1)) :- elim_equiv(X,X1), elim_equiv(Y,Y1).
elim_equiv((X => Y),(X1 => Y1)) :- elim_equiv(X,X1), elim_equiv(Y,Y1).
elim_equiv((X <=> Y),((X1 => Y1) /\ (Y1 => X1))) :- elim_equiv(X,X1),
        elim_equiv(Y,Y1).

elim_implic(X,X) :- atom(X).
elim_implic((~X),(~X1)) :- elim_implic(X,X1).
elim_implic((X \/ Y),(X1 \/ Y1)) :- elim_implic(X,X1), elim_implic(Y,Y1).
elim_implic((X /\ Y),(X1 /\ Y1)) :- elim_implic(X,X1), elim_implic(Y,Y1).
elim_implic((X => Y),((~X1) \/ Y1)) :- elim_implic(X,X1),
        elim_implic(Y,Y1).
```

```
get_nnf(F,F) :- is_nnf(F), !.
get_nnf((~(~X)),X1) :- get_nnf(X,X1).
get_nnf((~(X \/ Y)),(X1 /\ Y1)) :- get_nnf((~X),X1), get_nnf((~Y),Y1).
get_nnf((~(X /\ Y)),(X1 \/ Y1)) :- get_nnf((~X),X1), get_nnf((~Y),Y1).
get_nnf((X \/ Y),(X1 \/ Y1)) :- get_nnf(X,X1), get_nnf(Y,Y1).
get_nnf((X /\ Y),(X1 /\ Y1)) :- get_nnf(X,X1), get_nnf(Y,Y1).

is_nnf(X) :- atom(X).
is_nnf((~X)) :- atom(X).
is_nnf((X \/ Y)) :- is_nnf(X), is_nnf(Y).
is_nnf((X /\ Y)) :- is_nnf(X), is_nnf(Y).

or_rule([],_) :- !, fail.
or_rule([(X \/ Y)|T],[X,Y|T]).
or_rule([H|T],[H|T1]) :- or_rule(T,T1).

and_rule([],_,_) :- !, fail.
and_rule([(X /\ Y)|T],[X|T],[Y|T]).
and_rule([H|T],[H|T1],[H|T2]) :- and_rule(T,T1,T2).
```

Index

Bibliography

[1] Pietro Abate, Rajeev Goré, and Florian Widmann. Cut-Free Single-Pass Tableaux for the Logic of Common Knowledge. Manuscript. A short version was presented at the Workshop on Agents and Deduction, TABLEAUX 2007, Aix en Provence.

[2] Luca Alberucci. *The Modal μ-Calculus and the Logic of Common Knowledge*. PhD thesis, University of Bern, 2002.

[3] Robert Aumann. Agreeing to Disagree. *Annals of Statistics*, 4(6):1236–1239, 1976.

[4] Robert Aumann. Interactive Epistemology I: Knowledge. *Journal of Game Theory*, 28:263–300, 1999.

[5] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[6] Jon Barwise. Scenes and Other Situations. *Journal of Philosophy*, 78(7):369–397, 1981.

[7] Jon Barwise. Three Views of Common Knowledge. In *Proc. of the 2nd Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 365–379. Morgan-Kaufmann Publishers, 1988.

[8] Mordechai Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, 1990.

[9] Johan van Benthem, Jan van Eijck, and Barteld Kooi. Logics of Communication and Change. *Information and Computation*, 204:1620–1662, 2006.

[10] Johan van Benthen. "One is a Lonely Number": On the Logic of Communication. Technical report, ILLC, University of Amsterdam, 2002.

[11] Ivan Boh. *Epistemic Logic in the Latter Middle Ages*. Routledge, 1993.

[12] Ivan Bratko. *PROLOG Programming for Artificial Intelligence*. Addison-Wesley, 1986.

[13] Kai Brünnler and Martin Lange. Cut-Free Systems for Temporal Logic. *Journal of Logic and Algebraic Programming*, 76(2):216–225, 2008.

[14] Kai Brünnler and Thomas Studer. Syntactic Cut-Elimination for Common Knowledge. In *Proc. of Methods for Modalities M4M5*, pages 227–240, 2009.

[15] Yegor Bryukhov. Automatic Proof Search in Logic of Justified Common Knowledge. In *Proc. of the 4th Workshop "Methods for Modalities" (M4M'05)*, pages 187–201. Springer-Verlag, 2005.

[16] Robert Bull. Cut Elimination for Propositional Dynamic Logic Without *. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 38:85–100, 1992.

[17] Rudolf Carnap. The Journal of Symbolic Logic. *Journal of Symbolic Logic*, 11(2):33–46, 1946.

[18] Brian Chellas. *Modal Logic. An Introduction*. Cambridge University Press, 1980.

[19] Herbert Clark and Catherine Marshall. Definite Reference and Mutual Knowledge. In Aravind Joshi, Bonnie Webber, and Ivan Sag, editors, *Elements of Discourse Understanding*, pages 10–63. Cambridge University Press, 1981.

[20] Keith Clark. Negation as Failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[21] William Clocksin and Christopher Mellish. *Programmieren in Prolog (in German)*. Springer-Verlag, 1990.

[22] Helder Coelho and José Cotta. *Prolog by Example*. Springer-Verlag, 1988.

[23] Alain Colmerauer and Philippe Roussel. La naissance de Prolog (in French). draft of a paper in Thomas Bergin and Richard Gibson, editors, *History of Programming Languages*, ACM Press/Addison-Wesley, 1996.

[24] Jack Copeland. Meredith, Prior and the History of Possible Worlds Semantics. *Synthese*, 150(3):373–397, 2006.

[25] Yves Deville. *Logic Programming*. Addison-Wesley, 1990.

[26] Tony Dodd. *Prolog: a Logical Approach*. Oxford Science Publications, 1990.

[27] Ernest Allen Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Volume B: Formal Models and Semantics, pages 995–1072. Elsevier, Amsterdam, 1990.

[28] Ronald Fagin, Joseph Halpern, Yoram Moses, and Moshe Vardi. *Reasoning About Knowledge*. The MIT Press, Cambridge, MA, 1996.

[29] Robert Floyd. Assigning Meanings to Programs. In *Proc. of the American Mathematical Society Symposium on Applied Mathematics 19*, pages 19–31, 1967.

[30] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. of the 5th International Conference and Symposium*, pages 1070–1080, 1988.

[31] Lloyd Gerson. *Ancient Epistemology*. Cambridge University Press, 2009.

[32] Edmund Gettier. Is Justified True Belief Knowledge? *Analysis*, 23:121–123, 1963.

[33] Robert Goldblatt. *Logics of Time and Computation*, volume CSLI Lecture Notes 7. Center for the Study of Language and Information, 1987.

[34] Alvin Goldman. Discrimination and Perceptual Knowledge. *The Journal of Philosophy*, 73:771–791, 1976.

[35] Rajeev Goré. *Cut-Free Sequent and Tableau Systems for Propositional Normal Modal Logics*. PhD thesis, Cambridge University, 1992.

[36] Joseph Halpern and Yoram Moses. Knowledge and Common Knowledge in a Distributed Environment. *Journal of the ACM*, 37(3):549–587, 1990.

[37] Joseph Halpern and Yoram Moses. A Guide to Completeness and Complexity for Modal Logics of Knowledge and Belief. *Artificial Intelligence*, 54:311–379, 1992.

[38] David Harel, Dexter Kozen, and Jarzy Tyurin. *Dynamic Logic*. MIT Press, Cambridge, MA, 2000.

[39] Vincent Hendricks and John Symons. Where's the Bridge? Epistemology and Epistemic Logic. *Philosophical Studies*, 128:137–167, 2006.

[40] Jaakko Hintikka. *Logic and Belief. An Introduction to the Logic of the Two Notions*. King's College London Publications, 1962.

[41] Jaakko Hintikka. Individuals, Possible Worlds, and Epistemic Logic. *Noûs*, 1(1):33–62, 1967.

[42] Jaakko Hintikka. Reasoning about Knowledge in Philosophy: the Paradigm of Epistemic Logic. In Joseph Halpern, editor, *Proc. of the 1st Conference on Theoretical Aspects of Knowledge*, pages 63–80. Morgan-Kaufmann Publishers, 1986.

[43] Charles Antony Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[44] Ian Horrocks and Peter Patel-Schneider. Optimizing Description Logic Subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.

[45] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical Reasoning for Expressive Description Logics. In *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99), LNAI 1705*, pages 161–180. Springer Verlag, 1999.

[46] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical Reasoning for Very Expressive Description Logics. *Logic Journal of the IGPL*, 8(3):239–264, 2000.

[47] Karel Hrbáček and Thomas Jech. *Introduction to Set Theory*. CRC Press, 1999.

[48] Gerhard Jäger, Mathis Kretz, and Thomas Studer. Cut-Free Common Knowledge. *Journal of Applied Logic*, 5:681–689, 2007.

[49] Feliks Kluźniak and Stanisław Szpanowicz. *Prolog for Programmers*. Academic Press, 1985.

[50] Simo Knuuttila. *Modalities in Medieval Philosophy*. Routledge, 1993.

[51] Robert Kowalski. Predicate Logic as a Programming Language. In *Proc. of IFIP 74*, pages 569–574. North-Holland Publishing Company, 1974.

[52] Robert Kowalski. Algorithm = Logic + Control. *Comm. of the ACM*, 22(7):424–436, 1979.

[53] Saul Kripke. A Completeness Theorem in Modal Logic. *The Journal of Symbolic Logic*, 24(1):1–14, 1959.

[54] Saul Kripke. A Semantical Analysis of Modal Logic I: Normal Modal Propositional Calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[55] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantyne Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[56] Martin Lange and Colin Stirling. Focus Games for Satisfiability and Completeness of Temporal Logic. In *Proc. of the 16th Symposium on Logic in Computer Science (LICS'01)*, pages 357–365, 2001.

[57] Daniel Lehmann. Knowledge, Common Knowledge and Related Puzzles. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, pages 62–67, 1984.

[58] David Lewis. *Convention*. Blackwell Publishing, 2002.

[59] Orna Lichtenstein and Amir Pnueli. Propositional Temporal Logics: Decidability and Completeness. *Logic Journal of the IGPL*, 8(1):55–85, 2000.

[60] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[61] David Maier and David Warren. *Computing with Logic. Logic Programming with Prolog*. The Benjamin/Cummings Publishing Company, Inc., 1988.

[62] Wiktor Marek and Mirosław Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):587–618, 1991.

[63] John McCarthy. Circumscription: a Form of Nonmonotonic Reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[64] John McCarthy, Masahiko Sato, Takesh Hayashi, and Igarashi Shigeru. On the Model Theory of Knowledge. Technical Report STAN-CS-78-657, Stanford University, 1979.

[65] Drew McDermott. Nonmonotonic Logic II: Nonmonotonic Modal Theories. *Journal of the ACM*, 29(1):33–57, 1982.

[66] Drew McDermott and Jon Doyle. Non-Monotonic Logic I. *Artificial Intelligence*, 13:41–72, 1980.

[67] Robert Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 28:75–94, 1985.

[68] Sara Negri. Proof Analysis in Modal Logic. *Journal of Philosophical Logic*, 34:507–544, 2005.

[69] Hirokazu Nishimura. Semantical Analysis of Constructive PDL. *Publ. RIMS*, 18:427–438, 1982.

[70] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[71] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.

[72] Raymond Perrault and Philip Cohen. Elements of Discourse Understanding. In Aravind Joshi, Bonnie Webber, and Ivan Sag, editors, *It's For Your Own Good: a Note on Inaccurate Reference*, pages 217–230, 1981.

[73] John Pollock. *Contemporary Theories of Knowledge*. Rowman and Littlefield Publishers, 1986.

[74] Raymond Reiter. On Closed World Data Bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[75] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[76] Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[77] Neil Rowe. *Artificial Intelligence through Prolog*. Prentice-Hall, 1988.

[78] Stefan Schwendimann. A New One-Pass Tableau Calculus for PLTL. In *Proc. of the Int. Conference on Automated Reasoning with Analytic Tableaux and Related Methods, LNCS 1397*, pages 277–292. Springer-Verlag, 1998.

[79] David Steiner. *Belief Change Functions for Multi-Agent Systems*. PhD thesis, University of Bern, 2009.

[80] Leon Sterling and Ehud Shapiro. *The Art of Prolog. Advanced Programming Techniques*. The MIT Press, Cambridge, MA, 1986.

[81] Andrew Tanenbaum. *Modern Operating Systems*. Pearson Prentice-Hall, 2009.

[82] Anne Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2000.

[83] Frans Vorbraak. Generalized Kripke Models for Epistemic Logics. In Joseph Halpern, editor, *Proc. of the 4rd Conference on Theoretical Aspects of Knowledge*, pages 214–228. Morgan-Kaufmann Publishers, 1992.

[84] Yanjing Wang, Lakshmanan Kuppsamy, and Jan van Eijck. Verifying Epistemic Protocols under Common Knowledge. In *Proc. of the 12th Conf. on Theoretical Aspects of Rationality and Knowledge (TARK-2009)*, pages 257–266, 2009.

[85] Jan Wielemaker. An Overview of the SWI-Prolog Program Environment. In *Proc. of the 13th Int. Workshop on Logic Programming Environments*, pages 1–16. Tata Institute of Fundamental Research, Mumbai, 2003.

[86] Georg Henrik von Wright. An Essay in Deontic Logic and the General Theory of Action. *Acta Philosophica Fennica*, XXI, 1968.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname:	Ricardo Wehbe
Matrikelnummer:	05–131–479
Studiengang:	Informatik
	Bachelor ☐ Master ☐ Dissertation ☒
Titel der Arbeit:	Annotated Systems for Common Knowledge
Leiter der Arbeit:	Prof. Dr. G. Jäger

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Bern, September 14, 2010

.....
Ricardo Wehbe

Lebenslauf

1961	Geboren am 16. Mai in Río Cuarto, Argentinien.
1967 – 1973	Primarschule Río Cuarto.
1974 – 1978	Sekondarschule Río Cuart.
1979 – 1987	Ingenieurwissenschaftstudium an der Universität Córdoba, Argentinien.
1999 – 1992	MSc in Ingenieurwissenschaft an der Bundesuniversität Rio de Janeiro, Brasilien. Titel der Masterarbeit: *Verification of Concurrent Systems with Fixpoints*.
2005 –20	Doktorand bei Prof. Dr. G. Jäger an der Universität Bern.