# Software tool support for modular reasoning in modal logics of actions

Samuel Balco[1], Sabine Frittella[2], Giuseppe Greco[3], Alexander Kurz[1], and Alessandra Palmigiano[4,5]

[1] Department of Informatics, University of Leicester
[2] Laboratoire d'Informatique Fondamentale d'Orléans
[3] Department of Languages, Literature and Communication - Utrecht Institute of Linguistics
[4] Faculty of Technology, Policy and Management, Delft University of Technology
[5] Department of Pure and Applied Mathematics, University of Johannesburg

**Abstract.** We present a software tool for reasoning in and about propositional sequent calculi for modal logics of actions. As an example, we implement the display calculus D.EAK of dynamic epistemic logic. The tool generates embeddings of the calculus in the theorem prover Isabelle/HOL for formalising proofs about D.EAK. Integrating propositional reasoning in D.EAK with inductive reasoning in Isabelle/HOL, we verify the solution of the muddy children puzzle for any number of muddy children. There also is a set of meta-tools that allows us to adapt the software for a wide variety of user defined calculi.

## 1 Introduction

This paper is part of a long ranging project which aims at developing the proof theory of a wide variety modal logics of actions. The logical calculi to be developed should

- have 'good' proof theoretic properties,
- be built modularly from smaller calculi,
- have applications in a wide range of situations not confined to computer science.

In [22,19,18], we started from the observation that modal logics of actions such as propositional dynamic logic (PDL) or dynamic epistemic logic (DEL), despite having Hilbert style axiomatisations, typically did not have Gentzen systems with good proof theoretic properties. We found that a more expressive extension of sequent calculi, the *display calculi*, allow us to give a proof system D.EAK for a logic of action and knowledge, which enjoys the following properties that typically come with display calculi:

- a cut-elimination theorem that can be proved by instantiating a meta-theorem à la Belnap [7],
- modularity in the sense that—without endangering the cut-elimination theorem—connectives and rules can be added freely as long as one adheres to the 'display format'.

During this work we recognized that software tool support and interactive theorem provers will have to play an important role in our project in order to both

- perform proofs in which we reason in our calculi,
- formalise proofs about our calculi.

In this paper, we present a software tool providing the necessary infrastructure and an illustrating case study. The case study is of interest in its own right, even if Lescanne in [32] already provides a formalisation of the solution of the muddy-children puzzle. The propositional reasoning required is performed in the propositional calculus D.EAK and embedded into Isabelle in such a way that the necessary induction can be performed and verified in Isabelle.

**Contributions.** One aim of our software tool is to support research on the proof theory of modal logics of actions. The typical derivations may be relatively small, but they should be presented in a user interface in LaTeX in a style familiar to the working proof theorist. Moreover, in order to facilitate experimenting with different rules and calculi, meta-tools are needed that construct a calculus toolbox from a calculus description file.

Second aim is to support investigations into the question whether a calculus is suited to reasoning in some application area. To perform relevant case studies, one must deal with much bigger derivations and additional features such as abbreviations and derived rules are necessary. Another challenge is that applications may require additional reasoning outside the given calculus (e.g. induction), for which we provide an interface with the theorem prover Isabelle.

More specifically, in the work presented in this paper, we focus on D.EAK and aim for applications to epistemic protocols. In detail, we provide the following.

- A calculus description language that allows the specification of the terms/rules and their typesetting in ASCII, Isabelle and LaTeX in a calculus description file.
- A program creating from a calculus description file the calculus toolbox, which comprises the following.
  - A shallow embedding of the calculus in the theorem prover Isabelle. The shallow embedding encodes the terms and the rules of the calculus and allows us to verify in the theorem prover whether a sequent is derivable in D.EAK.
  - A deep embedding of the calculus in Isabelle. The deep embedding also has a datatype for derivations and allows us to prove theorems about derivations.
  - A user interface (UI) that supports
    - interactive creation of proof trees,
    - simple automatic proof search (currently only up to depth 5),
    - export of proof trees to LaTeX and Isabelle,
    - the use of derived rules, abbreviations, and tactics.
- A full formalisation of the proof system for dynamic epistemic logic of [22], which is the first display calculus of the logic of Baltag-Moss-Solecki [6] (without common knowledge).
- A fully formal proof of the solution of the muddy children puzzle for any number of dirty children. This is done by verifying in Isabelle that the solution of the muddy children puzzle can be derived in D.EAK for any number of muddy children.
- A set of meta-tools that enables a user to change the calculus.

**Case study: Muddy children** The muddy children puzzle was chosen because it is a well-known example of an epistemic protocol and required us to extend the tool from one supporting short proofs of theoretical value to larger proofs in an application domain. On the UI side, this led us to add features including abbreviations, macros (derived rules), and two useful tactics. On the Isabelle side, we added a shallow embedding of D.EAK in which we do the inductive proof that the well-known solution of the muddy children puzzle holds for arbitrary number of children. Whereas most of the proof is done in D.EAK using the UI and then automatically translating to Isabelle, the induction itself is based on the higher order logic of Isabelle/HOL.

**Related work.** The papers [13,14] pioneered the application of interactive theorem proving and Isabelle to the proof theory of display calculi of modal logic. A sequent calculus for dynamic epistemic logic was given by [15] and results on automatically proving correct the solution of the muddy children puzzle for small numbers of dirty children are reported in [39]. Tableau systems for automatic reasoning in epistemic logics with actions are studied in [1,2,35]. Even though tableaux are close relatives of sequent calculi, tableaux are designed towards efficient implementation whereas display calculi are designed to allow for a general meta-theory and uniform implementation. Their precise relationship needs further investigation in future work.

**Comparison of Isabelle to other proof assistants.** The papers [31,33,32] implement epistemic logic in the proof assistant Coq. It would be interesting to conduct the work of this paper based on Coq to enable an in-depth comparison. Isabelle has some advantages for us, in particular, (1) the proof language Isar, (2) the sledgehammer method, and (3) export of theories into programming languages such as Scala. This allows us to build the user interface (UI) directly on the deep embedding of the calculus in Isabelle, thus reusing verified code. This will be important to us in Section 4, where we use (1) and (2) in order to write the mathematical parts of the proof of the solution of the muddy children puzzle in a mathematical style close to [34] and we use (3) and the UI to build the derivations in D.EAK.

**Outline.** Section 2 reviews what is needed about D.EAK. Section 3 presents the main components of the DEAK calculus toolbox. Section 4 discusses the implementation of the muddy children puzzle. Section 5 explains the efforts we have made to keep the tool parametric in the calculus. Section 6 concludes with lessons learned and directions of future research.

**Acknowledgements.** At several crucial points, we profited from expert advice on Isabelle by Tom Ridge, Thomas Tuerk and Christian Urban. We thank Roy Crole and Hans van Ditmarsch for valuable comments on an earlier draft.

## 2 Display calculi and D.EAK

In this section, we will introduce Belnap's display calculi [7], which are a refinement of Gentzen sequent calculi. Having received widespread application (e.g. [30,43,23,9,12]), we will argue that display calculi form a good framework for systematically studying

modal logics. As a case study, we will also introduce a formalisation of the **D**isplay calculus for **E**pistemic **A**ctions and **K**nowledge (D.EAK) [22]; a display version of the the dynamic epistemic logic [42] of Baltag-Moss-Solecki [6] without common knowledge.

The current paper is part of a wider project that seeks to establish display calculi as a suitable framework for a wide variety of modal logics, both from the point of view of proof theory and from the point of view of tools supporting the reasoning in and about such logics. We use the D.EAK logic as a guiding example to show some of the features of the tools presented in the latter sections, but the tools and approaches presented in this paper do not depend on a detailed knowledge of display logics or D.EAK in particular. A complete description of D.EAK is available in [22] (where it is called D'.EAK).

**Display Calculi.** A tool or a framework for investigating modal logics in a systematic fashion should not only support as many of these logics as possible, it should also do this in a uniform way. That is, it should allow the user an easy way to define their own logic in a suitable calculus description language, as well as an easy way to combine already defined logics. After defining a new logic, either from scratch or by combining given logics, a reasoning tool for the new logic should be compiled automatically from the calculus description file.

Display calculi support such a modular and uniform approach in several ways. Firstly, the display format is restricted enough to allow us to define logics in a simple way: The language is given by a context free grammar and the inference rules by Horn clauses which only contain either logical connectives or their structural counterparts (and typically do not refer to external mechanisms such as side-conditions or labels encoding worlds in a Kripke model). Yet, display calculi are more *expressive* than sequent calculi (a wide array of logical properties in the language of display calculi is simply not expressible via Gentzen calculi [10]).

Secondly, the display format was invented because it supports a cut-elimination metatheorem, that is, any display calculus will enjoy cut-elimination. Intuitively, this is due to the meaning of a connective changing in a controlled way when the logic is extended, which in turn makes *modularity* useful. Adding or removing so-called analytic structural rules to a display calculus captures different logics, whilst still preserving cut elimination [11,25].

Finally, display calculi aim to be *transparent*: the design of inference rules corresponds to a programming discipline and leads to a well-understood class of algebraic, and via correspondence theory, Kripke semantics [11,25].

**D.EAK** is a proof system for (intuitionistic or classical) dynamic epistemic logic, the abstract syntax of formulas being defined by the grammar

$$\boxed{\phi ::= p \mid \bot \mid \top \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi \mid [\mathtt{a}]\phi \mid [\alpha]\phi \mid \langle \mathtt{a} \rangle \phi \mid \langle \alpha \rangle \phi \mid 1_\alpha} \tag{1}$$

where $p$ ranges over atomic propositions, $\mathtt{a}$ ranges over agents, with $[\mathtt{a}]\phi$ standing for "agent $\mathtt{a}$ knows $\phi$", and $\alpha$ ranges over actions with $[\alpha]\phi$ standing for "$\phi$ holds after $\alpha$". $1_\alpha$ represents the precondition of the action $\alpha$ in the sense of [6]. Negation is expressed by $\phi \to \bot$.

**Operational rules.** Display calculi are sequent calculi in which the rules follow a particular format that guarantees good proof theoretic properties such as cut elimination. One of the major benefits is modularity: different calculi can be combined and rules can be added while the good properties are preserved.

The rules of the calculus are formulated in such a way that, in order to apply a rule to a formula, the formula needs to be 'in display'. For example, the following or-introduction on the left (where 'contexts' are denoted by $W,X,Y,Z$ and formulas by $A,B$)

$$(\vee'_L)\quad \frac{W,A \vdash X \qquad Z,B \vdash Y}{W,Z,A \vee B \vdash X,Y} \tag{2}$$

is not permitted in a display calculus, since the formula $A \vee B$ must be introduced in isolation as, for example, in our rule

$$\boxed{(\vee_L)\quad \frac{A \vdash X \qquad B \vdash Y}{A \vee B \vdash X;Y}} \tag{3}$$

where $A$ and $B$ are formulas and $X,Y$ are arbitrary 'contexts'. Contexts are formalised below as 'structures', that is, as expressions formed from structural connectives. In sequent calculi, there is typically only one structural connective "," but display calculi generalise this. To emphasise that "," is now only one of many structural connectives we write it as ";".

**Display rules.** In order to derive a rule such as $(\vee'_L)$ from the rule $(\vee_L)$, it becomes necessary to isolate formulas by moving contexts to the other side. This is achieved by pairing the structural connectives such as ";" (written ';' in D.EAK) with so-called adjoint (aka residuated) operators such as ">" and adding bidirectional display rules

$$\boxed{(;,>)\ \frac{X;Y \vdash Z}{Y \vdash X > Z} \qquad \frac{Z \vdash X;Y}{X > Z \vdash Y}\ (>,;)}$$

which allow us to isolate, in this instance, $Y$ on the left or right of the turnstile.

The name display calculus derives from the requirement that the so-called display property needs to hold: Each substructure can be isolated on the left-hand side, or, exclusively, on the right-hand side. This is the reason why we can confine ourselves, without loss of generality, to the special form of operational rules discussed above.

**Structures.** A systematic way of setting this up for the set of formulas (1) is to introduce *structural connectives* corresponding to the operational connectives as follows.

| Structural | $<$ | | $>$ | | ; | | I | | $\{\alpha\}$ | | $\overbrace{\alpha}$ | | $\Phi_\alpha$ | | $\{\mathtt{a}\}$ | | $\overbrace{\mathtt{a}}$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Operational | $\prec$ | $\leftarrow$ | $\succ$ | $\rightarrow$ | $\wedge$ | $\vee$ | $\top$ | $\bot$ | $\langle\alpha\rangle$ | $[\alpha]$ | $\overbrace{\alpha}$ | $\underbrace{\alpha}$ | $1_\alpha$ | | $\langle\mathtt{a}\rangle$ | $[\mathtt{a}]$ | $\overbrace{\mathtt{a}}$ | $\underbrace{\mathtt{a}}$ |

This leads to a two tiered calculus which has formulas and structures, with structures generalising contexts and being built from structural connectives. We briefly comment on the particular choice of structural connectives above. Keeping with the aim of modularity, D.EAK was designed in such a way that one can drop the exchange rule for

';' and treat non-commutative conjunction and disjunction, in which case we need two adjoints of ';' denoted by $>$ and $<$. [6] Similarly, negation is formalised in terms of implication and bottom as in intuitionisitc and substructural logics. Following the symmetries inherent in this substructural analysis of logic [37] suggests to add the operational connectives $\prec$, $\leftarrow$, $\succ$ (but which are only needed if one does not have the rule of commutativity of ';'). Similarly, the modal operators $[\alpha]$ and $[\mathtt{a}]$ have structural counterparts $\{\alpha\}$ and $\{\mathtt{a}\}$ which in turn have adjoints $\overbrace{\alpha}$ and $\overbrace{\mathtt{a}}$. The formulas (1) do not have operational connectives corresponding to the structural connectives $\overbrace{\alpha}$ and $\overbrace{\mathtt{a}}$, but they can be added and are indeed useful (in terms of Kripke semantics, the adjoint of a box modality $\Box$ for a relation $R$ is the diamond modality for the converse relation $R^{-1}$ often denoted by $\blacklozenge$).

**Structural rules.** The rules of D.EAK can be divided into operational rules and display rules, as discussed above, and structural rules, to which we turn now. The operational rules such as $(\vee_L)$ specify how to introduce a logical operation. Display rules such as $(;>)$ are used to isolate formulas or structures to which we want to apply a specific rule. The logical axiomatisation sits in the structural rules. Apart from the structural rules like weakening, exchange, and contraction for ';' we have also other structural rules such as the display rules discussed above and rules that express properties such as 'actions are partial functions' axiomatised by the rule [7]

$$\boxed{\frac{X \vdash Y}{\{\alpha\}X \vdash \{\alpha\}Y}} \tag{4}$$

and such as 'if $\mathtt{a}$ knows $Y$, then $Y$ is true' axiomatised by the rule [8]

$$\boxed{\frac{X \vdash \{\mathtt{a}\}Y}{X \vdash Y}} \tag{5}$$

The reason to axiomatise a logic via such structural rules instead of axioms is that then cut-elimination is preserved.

**Modularity of D.EAK and related calculi.** We have seen that D.EAK has a large number of connectives. But they arise according to clear principles: operational connectives have structural counterparts which in turn have adjoints. Similarly, the fact that D.EAK as we defined it, due to being built systematically from substructural connectives, has a large number of rules does not pose problems from a conceptual point as the rules fall into clearly delineated classes each serving their own purpose. It is exactly this feature which enables the modularity of the display logic approach to the proof theory of sequent calculi. But, from the practical point of view of creating proof trees

[6] For example, taking into account the correspondence between operational and structural connectives, the rule $(;,>)$ above says precisely that the operation that maps $C$ to $A \to C$ is right-adjoint to the operation that maps $B$ to $A \wedge B$. Similarly, $(>,;)$ expresses that $A \succ \_$ is left adjoint to $A \vee \_$.

[7] which implies that one can derive $\langle\alpha\rangle X \vdash [\alpha]X$

[8] which implies that one can derive $[\mathtt{a}]Y \vdash Y$

or of composing a number of different calculi, this large number of connectives and rules makes working with these calculi difficult. Moreover, the encoding of terms and proof trees needed for automatic processing will not be readable to humans who would expect to manipulate proof trees displayed from LaTeX documents in an easy interactive way. How we propose to solve these problems will be discussed in the next section.

## 3 The D.EAK calculus toolbox

The aim of the D.EAK calculus toolbox[9] is to support research on the proof theory of dynamic epistemic logic as well as to conduct case studies exploring possible applications. It provides a shallow and a deep embedding of D.EAK into Isabelle and a user interface implemented in Scala.

The shallow embedding has an inductive datatype for the terms of the calculus and encodes the rules via a predicate describing which terms are derivable. It is used to prove correct the solution of the muddy children puzzle in Section 4.

The deep embedding also has datatypes for rules and derivations and provides functionality such as rule application (match and replace) as well as automatic proof search and tactics. The corresponding Isabelle code is exported to Scala and used in the user interface.

D.EAK proof trees can be constructed interactively in a graphical user interface by manipulating trees typeset in LaTeX. Proof trees can be exported to LaTeX/PDF and Isabelle. This was essential for creating the Isabelle proof in Section 4. Examples of typeset LaTeX proof trees can be found at [5]: The `.cs` files contain the proofs as done in the UI and the `.tex`-files the exported LaTeX code. The tag `cleaned_up` was added after a small amount of manual post-processing of the `.tex`-files.

### 3.1 Shallow embedding (SE) in Isabelle

The shallow embedding of the calculus D.EAK is available in the files `DEAK_SE.thy` and `DEAK_SE_core.thy`. The file `DEAK_SE_core.thy` contains the definitions of the terms via datatypes `Atprop, Formula, Structure, Sequent`. For example,

```
datatype Sequent = Sequent Structure Structure ("_ ⊢ _")
```

declares that an element of datatype `Sequent` consists of two structures. The annotation `("_ ⊢ _")` allows us to use the familiar infix notation $\vdash$ in the Isabelle IDE.

The file `DEAK_SE.thy` encodes the rules of the calculus by defining a predicate `derivable`

```
inductive derivable :: "Locale list ⇒ Sequent ⇒ bool"  ("_ ⊢d _")
```

by induction over the rules of D.EAK. For example, the rule $(\vee_L)$ above is encoded as

```
Or_L:  "l ⊢d (B ⊢ Y) ⟹ l ⊢d (A ⊢ X) ⟹ l ⊢d (A ∨ B ⊢ X ; Y)"
```

[9] Compiled version available for download at: https://github.com/goodlyrottenapple/calculus-toolbox/raw/master/calculi/DEAK.jar

which expresses in the higher-order logic of Isabelle/HOL that if $B \vdash Y$ and $A \vdash X$ are derivable, then $A \vee B \vdash X;Y$ is derivable. Note that $A, B, X, Y$ are variables of Isabelle. The rule will be applied using the built-in reasoning mechanism of Isabelle/HOL which includes pattern matching.

The datatype `Locale` is used to carry around all the information needed in a proof that is not directly available, in a bottom up proof search, from the sequent on which we want to perform a rule.

For example, in order to perform a cut, we need to specify the cut formula. In the UI, when constructing a prooftree interactively, it will be given by the user. Internally, cut-formulas are of type `Locale` and the cut-rule is given by

```
"(CutFormula f)∈set l ⟹ l⊢d(X ⊢ f) ⟹ l⊢d(f ⊢Y) ⟹ l⊢d(X ⊢Y)"
```

Similarly, the rules that describe the interaction of the knowledge of agents with epistemic actions depend on the so-called action structures, which define the actions, but are not part of the calculus itself. These action structures, therefore, are also encoded by data of type `Locale`.

Before coming to the deep embedding next, we would like to emphasise, that in order to prove in the shallow embedding, that a certain sequent is derivable in D.EAK, one shows in theorem prover Isabelle/HOL that the sequent is in the extension of the predicate `derivable`. The proof itself is not available as data that can be manipulated.

### 3.2 Deep embedding (DE) in Isabelle

The deep embedding is available in the files `DEAK.thy` and `DEAK_core.thy`. The latter contains the encoding of the terms of D.EAK, which differs only slightly from the one of the shallow embedding. It also contains functions match and replace, plus some easy lemmas about their behaviour. The functions match and replace are used in `DEAK.thy` to define how rules are applied to sequents.

`DEAK.thy` starts out by defining the datatypes `Rule` and `Prooftree`. The function `der` implements how to reason backwards from a goal:

```
fun der :: "Locale ⇒ Rule ⇒ Sequent ⇒ (Rule * Sequent list)"
```

takes a locale, a rule $r$, and a sequent $s$ and outputs the list of premises needed to prove the $s$ via $r$.[10] This function is then used to define the predicate `isProofTree` and other functions that are used by the UI.

One reason to define the deep embedding in Isabelle (and not e.g. directly in the UI) is that we want to use the deep embedding in future work, to implement and prove correct the cut elimination for D.EAK and related calculi. Another is that the UI then uses reliable code compiled from its specification in Isabelle.

[10] It is at this point where our implementation of the deep embedding is currently tailored towards substructural logics: For each rule $r$ and each sequent $s$, there is only one list of premises to consider. Generalising the deep embedding to sequent calculi with rules such as (2) would require a modification: If we interpret the structure $W, X, A \vee B$ in (2) not as a structure (ie tree) but as a list, then matching the rule (2) against a sequent would typically not determine the sublists matching $W$ and $X$ in a unique way. More information is available at [3].

### 3.3 Functionality of the user interface (UI)

The UI is an essential part of the toolbox and provides the following functionality:

- LaTeX typesetting of the terms of the calculus, with user specified syntactic sugar.
- Graphical representation of proof trees in LaTeX.
- Exporting proof trees to LaTeX/PDF and to Isabelle SE.
- Automatic proof search (to a modest depth of 5).
- Interactive proof tree creation and modification, including merging proof trees, deleting portions of proof trees, and applying rules.
- Tactics for deriving the generalised identity/atom rules and applying display rules.
- User defined abbreviations and macros (derived rules).

The UI is implemented in Scala. There were several reasons for choosing Scala, one of which is Isabelle's code export functionality which translates functions written in Isabelle theory files to be exported into functional languages such as Scala or Haskell, amongst others. This meant that the underlying formalisation of terms, rules and proof trees of the deep embedding of the calculus and the functions necessary for building and verifying proof trees could be built in Isabelle and then exported for the UI into Scala.

Another advantage of using Scala is the fact that it is based on Java and runs on the JVM, which makes code execution fast enough, and, more importantly, it is cross platform and allows the use of Java libraries. This was especially useful when creating the graphical UI for manipulating proof trees, as the UI depends on two libraries, JLaTeXMath and abego TreeLayout, which allow for easy typesetting and pretty-printing of the proof trees as well as simple visual creation and modification of proof trees in the UI.

The UI was made more usable with the implementation of some simple tactics, such as the display tactic, which simplifies the often tedious application of display rules to isolate a particular structure in a sequent. This happens in a display calculus every time the user wants to apply an operational rule, and can be a bureaucratic pain point of little interest to the logician. Even though these tactics are unverified, once the generated proof tree is exported to Isabelle, it will be checked by the Isabelle core to ensure validity.

## 4 Case study: The muddy children puzzle

The muddy children puzzle is a classical example of reasoning in dynamic epistemic logic, since it highlights how epistemic actions such as public announcements modify the knowledge of agents. We will recall the puzzle in some detail below. The solution will state that, after $k$ rounds of all agents announcing "I don't know", all agents do in fact know.

The correctness of the solution has been established, for all $k \in \mathbb{N}$ using induction, by informal mathematical proof [16] and by mathematical proofs about a formalisation in a Hilbert calculus [34]. It has also been automatically verified, for small values of $k$, using techniques from model checking [29] (see also [40] for related work) and automated theorem proving [15,39].

Here, we prove in Isabelle/HOL that for all $k$ the solution is derivable in D.EAK.

### 4.1 The muddy children puzzle

There are $n > 0$ children and $0 < k \leq n$ of them have mud on their foreheads. Each child sees (and hence knows) which of the others is dirty. But they cannot see (and therefore do not know at the beginning) whether they are dirty themselves (thus the number $n$ is known to them but $k$ is not). The first epistemic action is the father announcing (publicly and truthfully) that at least one of the children is dirty. From then on, the protocol proceeds in rounds. In each round, all children announce (simultaneously, publicly and truthfully) whether they know that they are dirty or not. How many rounds need to be played until the dirty children know that they are dirty?

In case $n = 1, k = 1$ the only child knows that it must be dirty, since the announcement by the father, as all announcements in this protocol, are assumed to be truthful. We write this as

$$[\mathsf{father}]\Box_1 D_1,$$

where $D_j$ is an atomic proposition encoding that child $j$ is dirty, $\Box_j p$ means child $j$ knows $p$ and $[\mathsf{father}]p$ means that $p$ holds after father's announcement. We use $\Box_j$ instead of $[j]$ for both stylistic purposes (the box $\Box$, being the usual notation in the literature), as well as to distinguish it from announcements like $[\mathsf{father}]$.

The case $n > 1$ and $k = 1$ is similar. Let $j$ be the dirty child. It sees, and therefore knows, that all the other children are clean. Since, after father's announcement, child $j$ knows that there is at least one dirty child, it must be $j$, and $j$ knows it.

In case $n > 1$ and $k = 2$ let $J = \{j, h\}$ be the set of dirty children. After father's announcement both $j$ and $h$ see one dirty child. But they do not know whether they are dirty themselves. So, according to the protocol, they announce that they do not know whether they are dirty. From the fact that $h$ announced $\neg\Box_h D_h$, child $j$ can conclude $D_j$, that is, we have $\Box_j D_j$. To see this, $j$ reasons that if $j$ was clean, then $h$ would be in the situation of the previous paragraph, that is, we had $\Box_h D_h$, in contradiction to the truthfulness of the announcement of $h$. Summarising, we have shown

$$[\mathsf{father}][\mathsf{no}]\Box_j D_j,$$

where $[\mathsf{no}]$ is the modal operator corresponding to the children announcing that they don't know whether they are dirty.

The cases for $k > 2$ follow similarly, so that we obtain for all dirty children $j$

$$[\mathsf{father}][\mathsf{no}]^{k-1}\Box_j D_j \tag{6}$$

For example, for $n = k = 100$, after 99 rounds of announcements "I don't know whether I am dirty" by the children, they all do know that they are dirty.

### 4.2 Muddy children in Isabelle

Our proof in Isabelle follows [34, Prop.24], which gives a mathematical proof that for all $n, k > 0$ there is, in a Hilbert system equivalent to D.EAK, a derivation of (6) from the assumption

$$\mathsf{dirty}(n, J) \wedge \mathsf{E}(n)^k(\mathsf{vision}(n)) \tag{7}$$

which encodes the rules of the protocol. Specifically, $\mathsf{dirty}(n,J)$ encodes for each $J \subseteq \{1,\ldots n\}$ that precisely the children $j \in J$ are dirty, $\mathsf{vision}(n)$ expresses that each child knows whether any of the other children are dirty, $\mathsf{E}(n)(\phi)$ means that 'every one of the $n$ children knows $\phi$' and $f^k$ indicates $k$-fold application of the function $f$ so that $\mathsf{E}(n)^k(\mathsf{vision}(n))$ says that 'each child knowing whether the others are dirty' is common knowledge up to depth $k$.

This means that we need to prove by induction on $n$ and $k$ that for all $n,k$ there is a derivation in the calculus D.EAK of the sequent

$$\mathsf{dirty}(n,J), \mathsf{E}(n)^k(\mathsf{vision}(n)) \vdash [\mathsf{father}][\mathsf{no}]^{k-1} \Box_j D_j \; . \tag{8}$$

where the actions father and no also depend on the parameter $n$.

For the cases $k = 1,2$ the proofs can be done with a reasonable effort in the UI of the tool, filling in all the details of the proof of [34].

But as a propositional calculus, D.EAK does not allow us to do induction. Therefore we use the shallow embedding of D.EAK and do the induction in the logic of Isabelle. The expressions $\mathsf{dirty}(n,J)$ and $\mathsf{E}(n)^k(\mathsf{vision}(n))$ and $[\mathsf{father}][\mathsf{no}]^{k-1} \Box_j D_j$ then are Isabelle functions that map the parameters $n,k$ to formulas (in the shallow embedding) of D.EAK, see the theory `MuddyChildren` [5].

The first part of the theory `MuddyChildren` contains the definitions of the formulas discussed above and establishes some of their basic properties. The actual proof is given as lemma `dirtyChildren`. We have taken care to follow [34] closely, so that the proof of its Proposition 24 can be read as a high-level specification of the proof in Isabelle of lemma `dirtyChildren`.

The proof in `MuddyChildren` differs from its specification in [34] only in a few minor ways. Instead of assuming the axiom of introspection $[\mathrm{a}]p \to p$, we added the corresponding structural rules to the calculus. This seems justified as it is a fundamental property of knowledge we are using and also illustrates a use of modularity. Instead of introducing separate atomic propositions for dirty and clean, we treat clean as an abbreviation for not dirty, which relieves us from axiomatising the relationship between dirty and clean explicitly. But if we want an intuitionistic proof, we need to add to our assumptions that 'not not dirty' implies dirty.

### 4.3 Conclusions from the case study

It took approximately 4 person-weeks to implement the proof of [34, Prop.24] in Isabelle. Part of this went into providing some 'infrastructure' contained in the theories `NatToString` and `DEAKDerivedRules` that could be reused for other case studies. On the other hand, we should say that it took maybe half a year to learn Isabelle and we couldn't have learned it from documentation and tutorials alone. At crucial points we had expert advice by Thomas Tuerk, Tom Ridge and Christian Urban.

For the construction of the proof in Isabelle, we made extensive use of the UI. Large parts of the Isabelle proof were constructed in the UI and exported to Isabelle.

One use one can make of the formal proof is to investigate which proof principles are actually needed. For example, examining the proof in `MuddyChildren`, it is easy to establish that the only point where a non-intuitionistic principle is used is to prove $\neg\neg D_j \to D_j$. Instead we could have added this formula (which only says that "not clean implies dirty") to the logical description of the puzzle (7).

It may be worth pointing out that this analysis is based on the substructural analysis of classical logic on which D.EAK is built. Accordingly, a proof in D.EAK is intuitionistic if and only if it does not use the so-called Grishin rules `Grishin_L` and `Grishin_R` (as defined in `DEAK.json`). Thus a simple text search for 'Grishin' in the theory `MuddyChildren` suffices to establish the claim that adding $\neg\neg D_j \to D_j$ to the assumptions, our proof of the muddy children puzzle follows the principles of intuitionistic logic.

## 5 *Meta-toolbox* - Building your own calculus toolbox

As discussed in Section 3, the D.EAK toolbox consists of a set of Isabelle theory files that formalize the terms and rules of this calculus, providing a base for reasoning about the properties of the calculus in the Isabelle theorem prover. The toolbox also includes a UI for building proof trees in the calculus.

As this paper is part of a wider program of the study of modal logics, the reader might naturally be interested in building her own calculus/logic, either building on top of D.EAK or starting from scratch. To do this, we provide a *meta-toolbox*, which consists of a set of scripts and utilities used for maintaining, modifying and building your own calculus toolbox (we will use italics when referring to the *meta-toolbox*, to avoid confusion with the toolbox generated by the *meta-toolbox* for a specific logic).

The *meta-toolbox* supports building tools for a wide range of propositional modal logics. Due to the display framework, constraints placed on the shape of the calculus terms and rules allow us to build large portions of the Isabelle theory files and the UI front-end in a generic, logic-agnostic way, from a calculus description file.

However, there are cases in which the language used to specify calculus description files may not be expressive enough to encode certain information about the logic. For example, in the case of rules with side-conditions, the user needs to implement the Isabelle code handling the side conditions manually. To avoid reimplementation on each change of the calculus description file, we provide the watcher utility, which builds a new calculus toolbox by weaving the user-specific manual changes into the generic automatically created code without breaking the user made modifications.

The main component of the *meta-toolbox* is the build script, which takes in a description file and expands it into multiple Isabelle theories and Scala code. Due to this centralised definition of the calculus, adding rules or logical connectives becomes easy as the user only needs to change the calculus description file and does not need to worry about how these changes affect multiple Isabelle and Scala files. The *meta-toolbox* thus allows for a more structured and uniform maintenance of the different encodings along with the UI. A detailed documentation and a tutorial is available [3].

### 5.1 Describing a calculus

We highlight some elements of how to describe a calculus such as D.EAK in the format that can be read by the *meta-toolbox*.

The calculus is described in a *calculus description file* using the JavaScript Object Notation (JSON), in our example `DEAK.json`. This file specifies the types (Formula, Structure, Sequent, ...), the operational and structural connectives, and the rules. For example, linking up with the discussion in Section 3, in

```
"Sequent": {
  "type" : ["Structure", "Structure"],
  "isabelle" : "_ \\<turnstile> _",
  "ascii" : "_ |- _",
  "latex" : "_ {\\ {\\textcolor{magenta}\\boldsymbol{\\vdash}\\ } _",
  "precedence": [311,311,310]
}
```

`"type"` specifies that a sequent consists of two structures.[11] The next three lines specify how sequents will be typeset in Isabelle, ASCII and LaTeX. To make proofs readable in the UI, the user can specify bespoke sugared notation using, for example, LaTeX commands such as colours and fonts.

Next we explain how rules are encoded. The encoding is divided into two parts. In the first part, under the heading `"calc_structure_rules"` the rules are declared. For example, we find

```
"Or_L" : {
  "ascii" : "Or_L",
  "latex" : "\\vee_L"
}
```

telling us how the names of the rule are typeset in ASCII and LaTeX. The rule (3) itself is described in the second part under the heading `"rules"` by

```
"Or_L" : ["F?A \\/ F?B |- ?X ; ?Y", "F?A |- ?X", "F?B |- ?Y"]
```

the first sequent of which is the conclusion, the following being the premises of the rule. The `?` has been defined in `DEAK.json` to indicate the placeholders (a.k.a. free variables or meta-variables) that are instantiated when applying the rule. The `F` marks placeholders that can be instantiated by formulas only.

The description of `Or_L` above suffices to compile it to Isabelle. But some rules of D.EAK need to be implemented subject to restrictions expressed separately. For example, the so-called atom rule formalises that in D.EAK, actions do not change facts (but they may change knowledge). Thus, whereas the rule is encoded as

```
"Atom" : ["?X |- ?Y", ""]
```

[11] The presence of the `\\` instead of just one `\` is unfortunate but `\` is a reserved character that needs to be escaped using `\`.

we need to enforce the condition that `?X |- ?Y` is of the form $\Gamma p \vdash \Delta p$, where $p$ is an atomic proposition and $\Gamma, \Delta$ are strings of action modalities. This is done by noting in the calculus description file the dependence on a condition called `atom` as follows.

```
"Atom" : {
  "condition" : "atom"
}
```

The condition `atom` itself is then implemented directly in Isabelle.

For bottom-up proof search, the deep embedding provides a function that, given a sequent and a rule, computes the list of premises (if the rule is applicable). For the cut rule, this is implemented by looking for a cut-formula in the corresponding `Locale`, see Section 3.1. (As stated earlier, whilst the calculus admits a cut free presentation, it is nonetheless useful to keep the cut rule in the calculus when manually constructing proofs.)

```
"RuleCut" : {
  "SingleCut" : {
    "locale" : "CutFormula f",
    "premise" : "(\\<lambda>x. Some [((?\\<^sub>S ”X”) ...",
    "se_rule" : "l \\<turnstile>d (X \\<turnstile>\\<^sub>S f ..."
  }
}
```

After `"premise"` we find the Isabelle definition of the DE-version of the rule and after `"se_rule"` the SE-version of the rule.

The most complicated rules of D.EAK are those which describe the interaction of action and knowledge modalities and we are not going to describe them here. They need all of the additional components `condition`, `locale`, `premise`, `se_rule`, to deal with side conditions which depend on actions being agent-labeled relations on actions.

The ability to easily change the calculus description file will be useful in the future, but also appeared already in this work. Compared to the version of D.EAK from [22], we noticed during the work on the muddy children puzzle that we wanted to add rules `Refl_ForwK` expressing $[a]p \to p$ (i.e. that the knowledge-relation is reflexive) and rules `Pre_L` and `Pre_R` allowing us to replace in a proof the constant representing the precondition of an action by the actual formula expressing the precondition. Using the *meta-toolbox*, this change was a simple case of adding the rule to the JSON description file and recompiling the calculus.

### 5.2 The build script, the template files, and the watcher utility

To build the calculus toolbox from the calculus description file `DEAK.json`, one runs the Python script, passing the description file to the script via the `--calculus` flag. This produces the Isabelle code for the shallow and deep embedding and the Scala code for the UI. By default, this toolbox is output to a directory called `gen_calc`.

**Template files.** The toolbox is generated from both the calculus description file and template files. Template files contain the code that cannot be directly compiled from the calculus description file, for example, the code of the UI. But whereas the code of the UI, in the folder `gui`, is independent of the particular calculus, the parser `Parser.scala` and the print class `Print.scala` consist of code written by the developer as well as code automatically generated from the calculus description file. Similarly, whereas parts of `DEAK.thy` are compiled from the calculus description file, other parts, such as the lemmas and their proofs are written by the developer.

**The Isabelle and Scala builder.** In order to support the weaving of automatically generated code into the template files, there are two domain specific languages defined in the files `isabuilder.py` and `scalabuilder.py`. For example, in the template file `Calc_core.thy`, from which `DEAK.thy` is generated, the line

```
(*calc_structure*)
```

prompts the build script to call a method defined in `isabuilder.py` which inserts the Isabelle definition of the terms of the calculus into `DEAK.thy`.

**The watcher utility.** In order to make the maintenance of the template files easier there is a watcher utility which allows, instead of directly modifying the template files, to work on the generated code. For example, if we want to change how proof search works, we would make the changes to the Isabelle file `DEAK.thy` and not directly to the template file `Calc_core.thy`. The watcher utility, when launched, runs in the background and monitors the specified folder. Any changes made to a file inside this folder are registered and the utility decompiles this file back into its corresponding template, each time a modification occurs. The watcher utility decompiles a file by looking for any sections of the file that have been automatically generated, and replacing these definitions by the special comments that tell the build script where to put the auto-generated code. In order for the decompiling to work correctly, the auto-generated code must be enclosed by special delimiters. Looking back at the example of `(*calc_structure*)`, when the template file is processed by the build script and expanded with the definitions from a specific calculus description file, the produced code is enclosed by the following delimiters:

```
(*calc_structure-BEGIN*)
auto-generated code ...
(*calc_structure-END*)
```

Hence, when the watcher utility decompiles a file into a template, it simply replaces anything of the form `(*<identifier>-BEGIN*) ... (*<identifier>-END*)` by the string `(*<identifier>*)`.

## 6 Conclusion

We presented a software tool that makes interactive theorem proving available for the proof theoretic study of modal logics of actions. From a calculus description file, shallow and deep embeddings of deductions are generated. The deep embedding is used

to automatically generate verified code for the user interface, which in turn allows us to make derivations in the calculus in a familiar proof theoretic environment and then export it to Isabelle. This has been used to develop a fully formalised proof of the correctness of the solution for the muddy children puzzle, making use of Isabelle's ability of inductive reasoning that goes beyond the expressiveness of (propositional) modal logic.

An interesting lesson learned from using interactive theorem proving in the proof theory of modal logics is that in our work the concerns of the software engineer and the proof theorist can be seen as two sides of the same coin as we will explain in the following.

From the point of view of proof theory, we are interested in developing 'good' calculi, which refers to, on the one hand, mathematical properties such as cut-elimination, and, on the other hand, to the design criteria developed in the area of proof theoretic semantics [38,36]. These design criteria include the following. (i) The meaning of a connective should be defined, in the spirit of introduction and elimination rules, in a way that renders their meaning independent of what other connectives and rules may be added to the calculus. (ii) The rules should be closed under uniform substitution and be free from extra-logical labels and side-conditions. (See (3) for an example.)

From a software engineering point of view, we want to (I) build software for a 'big' logical calculus comprising many connectives in a modular way, connective by connective and (II) provide the user with a domain specific language that allows for a user-friendly specification of a calculus and its rules. In particular, it should be possible to automatically build a set of tools that allows high-level reasoning and implementation independent use of the calculus from a single calculus description file.

A lesson learned is that (i) and (I) as well as (ii) and (II) are closely related. While we admit that our domain specific language is rudimentary and the calculus description files in Section 5.1 could be much more user-friendly, the main issues that need further research are extra-logical labels and side-conditions, see e.g. page 14 where we write "The condition `atom` itself is then implemented directly in Isabelle". Indeed, such side-conditions are not easily formulated without knowing the lower-level implementation of the logics (in our case their implementation in Isabelle) and therefore are in conflict with the software engineering principle of shielding the user from implementation details. While we solved this problem using a software engineering method (see Section 5.2), the next paragraph discusses a possible proof-theoretic solution, namely multi-type display calculi.

The move to multi-type display calculi is akin to the move from one-sorted algebras to many-sorted algebras. Multi-type display calculi, introduced in [19], allow us to absorb extra-logical labels and side conditions into the types. For example, by introducing a type for atoms, the condition on substitution becomes uniform substitution (of formulas of the correct type). Similarly, the extra-logical labels needed for actions can be eliminated. (In passing, we also note that the well-known side conditions for the rules of first-order quantifiers can be eliminated in this way.) This methodology has, by now, been successfully applied to a range of calculi [27,26,20,21,28,25].

Ongoing and future work arises from the discussion above. First, a new *meta-toolbox* for multi-type sequent and display calculi (an alpha version providing a more user-friendly interface to define calculi and manipulate derivations is already available [4]). Second, following the work of [13,14] on proving cut elimination of display calculi, a full formalisation of cut-elimination for D.EAK, or rather of the cut-elimination meta-theorems of [22,19,18]. Third, integrating interactive theorem proving with automatic proof search, much in the spirit of Isabelle's Sledgehammer. In particular, modal logics of actions can have tableau systems that do efficient automatic proof search [1]. One question here is, whether it will be possible to do this integration in a modular way: In the light of our discussion above, tableau systems are closely related to sequent calculi [17,24], but they typically do not do so well w.r.t. to property (II).

## References


1. G. Aucher and F. Schwarzentruber. On the complexity of dynamic epistemic logic. In *Proceedings of the 14th Conference on Theoretical Aspects of Rationality and Knowledge* (TARK 2013).
2. P. Balbiani, H. van Ditmarsch, A. Herzig, and T. de Lima. Tableaux for public announcement logic. *Journal of Logic and Computation*, 20(1):55–76, 2010.
3. S. Balco. The calculus toolbox. Download and documentation at https://goodlyrottenapple.github.io/calculus-toolbox/.
4. S. Balco. The calculus toolbox 2. Download and documentation at https://github.com/goodlyrottenapple/calculus-toolbox-2.
5. S. Balco and S. Frittella. Muddy children in Isabelle. https://goodlyrottenapple.github.io/muddy-children/.
6. A. Baltag, L. S. Moss, and S. Solecki. The logic of public announcements, common knowledge and private suspicious. Technical Report SEN-R9922, CWI, Amsterdam, 1999.
7. N. Belnap. Display logic. *Journal of Philosophical Logic*, 11:375–417, 1982.
8. P. Blackburn, J. van Benthem, and F. Wolter, editors. *Handbook of Modal Logic*. Elsevier, 2006.
9. J. Brotherston. Bunched logics displayed. *Studia Logica*, 100(6):1223–1254, 2012.
10. A. Ciabattoni, N. Galatos, and K. Terui. From axioms to analytic rules in nonclassical logics. In *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science* (LICS 2008).
11. A. Ciabattoni and R. Ramanayake. Power and limits of structural display rules. *ACM Transactions on Computational Logic (TOCL)*, 17(3):17, 2016.
12. A. Ciabattoni, R. Ramanayake, and H. Wansing. Hypersequent and display calculi - a unified perspective. *Studia Logica*, 102(6):1245–1294, 2014.
13. J. E. Dawson and R. Goré. Embedding Display Calculi into Logical Frameworks: Comparing Twelf and Isabelle. *Electr. Notes Theor. Comput. Sci.*, 42:89–103, 2001.
14. J. E. Dawson and R. Goré. Formalised cut admissibility for display logic. In *Theorem Proving in Higher Order Logics, 15th International Conference*, TPHOLs 2002.
15. R. Dyckhoff, M. Sadrzadeh, and J. Truffaut. Algebra, proof theory and applications for an intuitionistic logic of propositions, actions and adjoint modal operators. *ACM Transactions on Computational Logic*, 14(4), 2013.
16. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
17. M. Fitting. *Proof Methods for Modal and Intuitionistic Logic*. Springer, 1983.
18. S. Frittella, G. Greco, A. Kurz, and A. Palmigiano. Multi-type display calculus for propositional dynamic logic. *J. Log. Comput.*, 26(6):2067–2104, 2016.
19. S. Frittella, G. Greco, A. Kurz, A. Palmigiano, and V. Sikimic. Multi-type display calculus for dynamic epistemic logic. *J. Log. Comput.*, 26(6):2017–2065, 2016.
20. S. Frittella, G. Greco, A. Kurz, A. Palmigiano, and V. Sikimić. Multi-type sequent calculi. In M. Z. Andrzej Indrzejczak, Janusz Kaczmarek, editor, *Trends in Logic XIII*, pages 81–93. Lodź University Press, 2014. https://arxiv.org/abs/1609.05343.
21. S. Frittella, G. Greco, A. Palmigiano, and F. Yang. A multi-type calculus for inquisitive logic. WoLLIC 2016.
22. S. Frittella, G. Greco, A. Kurz, A. Palmigiano, and V. Sikimic. A proof-theoretic semantic analysis of dynamic epistemic logic. *J. Log. Comput.*, 26(6):1961–2015, 2016.
23. R. Goré. Substructural logics on display. *Logic Journal of IGPL*, 6(3):451–504, 1998.
24. R. Goré. Tableau methods for modal and temporal logics. In M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga, editors, *Handbook of Tableau Methods*. Springer, 1999.

25. G. Greco, M. Ma, A. Palmigiano, A. Tzimoulis, and Z. Zhao. Unified correspondence as a proof-theoretic tool. *J. Log. Comput.*, doi: 10.1093/logcom/exw022, 2016.
26. G. Greco and A. Palmigiano. Linear logic properly displayed. *CoRR*, abs/1611.04181, 2016.
27. G. Greco and A. Palmigiano. Lattice logic properly displayed. WoLLIC 2017.
28. G. Greco, F. Liang, A. Moshier, and A. Palmigiano. Multi-type display calculus for semi De Morgan logic. WoLLIC 2017.
29. J. Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving: A manifesto. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 325–334, 1991.
30. M. Kracht. Power and weakness of the modal display calculus. In *Proof theory of modal logic*, pages 93–121. Kluwer, 1996.
31. P. Lescanne. Mechanizing common knowledge logic using COQ. *Ann. Math. Artif. Intell.*, 48(1-2):15–43, 2006.
32. P. Lescanne. Common knowledge logic in a higher order proof assistant. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 271–284, 2013.
33. P. Lescanne and J. Puisségur. Dynamic logic of common knowledge in a proof assistant. *CoRR*, abs/0712.3146, 2007.
34. M. Ma, A. Palmigiano, and M. Sadrzadeh. Algebraic semantics and model completeness for intuitionistic public announcement logic. *Annals of Pure and Applied Logic*, 165(4):963–995, 2014.
35. M. Ma, K. Sano, F. Schwarzentruber, and F. R. Velázquez-Quesada. Tableaux for non-normal public announcement logic. In *Logic and Its Applications - 6th Indian Conference, ICLA 2015, Mumbai, India, January 8-10, 2015. Proceedings*, pages 132–145, 2015.
36. T. Piecha and P. Schroeder-Heister, editors. *Advances in Proof-Theoretic Semantics*. Springer, 2016.
37. G. Restall. *An Introduction to Substructural Logics*. Routledge, London, 2000.
38. P. Schroeder-Heister. Proof-theoretic semantics. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition, 2016.
39. J. Truffaut. Implementation and improvements of a cut-free sequent calculus for dynamic epistemic logic, 2011. MSc thesis, University of Oxford.
40. H. van Ditmarsch, J. van Eijck, I. Hernández-Antón, F. Sietsma, S. Simon, and F. Soler-Toscano. Modelling cryptographic keys in dynamic epistemic logic with DEMO. In *10th International Conference on Practical Applications of Agents and Multi-Agent Systems*, PAAMS 2012.
41. H. P. van Ditmarsch and B. Kooi. *One Hundred Prisoners and a Light Bulb*. Springer, 2015.
42. H. P. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic Epistemic Logic*. Springer, 2007.
43. H. Wansing. *Displaying Modal Logic*. Kluwer, 1998.