2016

Tool Support for Reasoning in Display Calculi

Samuel Balco
University of Oxford

Sabine Frittella
Delft University of Technology

Giuseppe Greco
Delft University of Technology

Alexander Kurz
Chapman University, akurz@chapman.edu

Alessandra Palmigiano
Delft University of Technology

Tool support for reasoning in display calculi

Samuel Balco[1⋆], Sabine Frittella[2], Giuseppe Greco[2], Alexander Kurz[3], and Alessandra Palmigiano[24]

[1] Department of Computer Science, University of Oxford
[2] Faculty of Technology, Policy and Management, Delft University of Technology
[3] Department of Computer Science, University of Leicester
[4] Department of Pure and Applied Mathematics, University of Johannesburg

Abstract. We present a tool for reasoning in and about propositional sequent calculi. One aim is to support reasoning in calculi that contain a hundred rules or more, so that even relatively small pen and paper derivations become tedious and error prone. As an example, we implement the display calculus D.EAK of dynamic epistemic logic. Second, we provide embeddings of the calculus in the theorem prover Isabelle for formalising proofs about D.EAK. As a case study we show that the solution of the muddy children puzzle is derivable for any number of muddy children. Third, there is a set of meta-tools, that allows us to adapt the tool for a wide variety of user defined calculi.

1 Introduction

Applied logic. An important development in logic, and in particular of logic in computer science, has been the move away from logic to logics. The need for automation of reasoning in AI or verification led to the design of hundreds of bespoke logics with good algorithmic properties for particular tasks. This development is particularly conspicuous in modal logic, the classic instance of which, Kripke's modal logic **K**, is decidable just because it is a certain fragment of first-order logic [8]. To compensate for the lack of expressiveness that comes with decidability, one develops modal logics focussed on different aspects such as time, probability, space, etc. See [9] for an overview of examples and techniques.

The proof theory of modal logic has had many successes, for example the tableaux methods of description logics with its applications to knowledge representation and ontologies [14,22,1]. Nevertheless, for many of the more sophisticated modal logics good proof systems are not known. A notable exception is the cut free sequent calculus of [12] for dynamic epistemic logic (without common knowledge). But given the diversity of modal logics and the importance of proof systems for reasoning about applications, it is desirable to have a systematic and uniform approach to the construction of modal proof systems with good proof theoretic properties.

Display calculi. Following work on so-called display calculi [7,23,18,31,19], we engaged in the systematic study of display calculi of dynamic modal logics in [17,16,15].

⋆ Supported by an EPSRC-funded Vacation Bursary in Summer 2014 and the University of Leicester Career Development Service Graduate Gateways programme in Summer 2015.

The principal advantage of display calculi is that they are built in a modular way and important proof theoretic properties such as cut-elimination are preserved under combination of logics (if the combined logic is still displayable).

Dynamic Epistemic Logic. As a case study for the feasibility of this approach we developed a display calculus D.EAK for the dynamic epistemic logic [30] of Baltag-Moss-Solecki [6] without common knowledge. On the one hand this logic contains features that are a challenge from a proof theoretic point of view. On the other hand, dynamic epistemic logic has many applications, both in computer science and in other areas. One particular interest is the verification of security protocols that involves epistemic notions. In this paper, as a case study of intermediate complexity, we give a full proof of the muddy children puzzle. A nice collection of more epistemic puzzles is available in [29].

Modularity. One aspect of the modularity of display calculi is that the logic is axiomatised by the structural rules, which can be added or removed in a flexible way. For example, even though D.EAK is based on classical propositional logic it can as well be based on intuitionistic or substructural logics by removing some of the structural rules. Another aspect of modularity is that it is easy to combine different such display calculi. But modularity also comes at cost. For example, D.EAK has more than a hundred rules. This poses no conceptual problems as the space of rules is well structured according to clear proof theoretic principles, but it does pose the practical problem of conducting the proofs and writing them down without making mistakes. Consequently, already for proof theoretic studies alone, tool support will be valuable.

Contributions. One aim of the tool is to support researchers working on the proof theory of display calculi. The typical derivations may by relatively small, but they must 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 given calculus description file.

The 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, 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 and rules as well as of 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 [17], which is the first display calculus of the logic of Baltag-Moss-Solecki [6] (without common knowledge).
- A fully formal proof (implemented in Isabelle) of the muddy children puzzle.
- A set of meta-tools that enables a user to change the calculus.

Case study: Muddy children The first version of the tool presented at ALCOP 2015 supported the interactive construction of the proof trees in [17] and their output to LaTeX. These proofs are not longer than a few dozen steps and of interest to establish the mathematical result of completeness of the D.EAK.

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, we added 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 [10,11] on proving cut elimination of display calculi in Isabelle have been a source of inspiration. Indeed, proving in Isabelle the variations [17,16,15] of Belnap's cut-elimination theorem remains one of our aims and we consider what we present in this paper as a necessary first step: Due to the notational overhead resulting from encoding the mathematical description of D.EAK into its Isabelle formalisation, we found constructing even the simplest derivations in Isabelle too burdensome without tool support.

The papers [24,26,25] implement epistemic logic in the proof assistant Coq. It would be very interesting to conduct the work of this paper based on Coq to enable an in depth comparison.

Comparison of Isabelle to other proof assistants. Isabelle has the following advantages for us.

1. Isabelle supports the proof language Isar supporting a style of writing proofs that is close to mathematical practice.
2. Isabelle provides the so-called sledgehammer method, which uses specialised automatic theorem provers that are able to discharge much of the tedious, low level reasoning.

3. Isabelle can export theories into programming languages such as Scala. This allows us to build the user interface directly on the deep embedding of the calculus in Isabelle, thus reusing verified code.

As far as we know Isabelle is the only proof assistant featuring all of the above. 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 [27] and we use (3) and the user interface to build the derivations in D.EAK.

Outline. Section 2 reviews what one needs to know 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 discusses directions of future research we plan to pursue.

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 The display calculus D.EAK

This section gives some background on D.EAK; for a complete description we refer to [17] (where it is called D'.EAK).

Formulas. D.EAK is a proof system for (intuitionistic or classical) dynamic epistemic logic the formulas of which are defined by induction as follows:

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

where $\mathtt{a}$ ranges over agents, with $[\mathtt{a}]\phi$ standing for "agent $\mathtt{a}$ knows ϕ", and α ranges over actions with $[\alpha]\phi$ standing for "ϕ holds after α". 1_α represents the precondition of the action α 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

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

Display rules. In order to derived 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

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

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

The name display calculus derives from the requirement that in a display calculus 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		$\{\alpha\}$		$\overbrace{\alpha}$		Φ_α		$\{\mathtt{a}\}$		$\overbrace{\mathtt{a}}$	
Operational	$\prec\!\!\!-$	$\leftarrow$	$-\!\!\!\succ$	$\rightarrow$	$\wedge$	$\vee$	$\top$	$\bot$	$\langle\alpha\rangle$	$[\alpha]$	$\overbrace{\alpha}$	$\underbrace{\alpha}$	1_α		$\langle\mathtt{a}\rangle$	$[\mathtt{a}]$	$\overbrace{\mathtt{a}}$	$\underbrace{\mathtt{a}}$

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. This means that we need two adjoints of ';' denoted by $>$ and $<$. [5] Following the symmetries inherent in this substructural analysis of logic suggests to add the operational connectives $\prec\!\!\!-$, $\leftarrow$, $-\!\!\!\succ$, but they are not needed in the following. 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

[5] 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 $A\vee\_$.

rules such as the display rules discussed above and rules that express properties such as 'actions are partial functions' axiomatised by the rule

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

and such as 'if a knows Y, then Y is true' axiomatised by

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

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 has over a hundred rules poses no 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 a practical point of view of creating proof trees 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 latexed prooftrees in an easy interactive way. How we propose to solve these problems will be discussed in the next section.

3 The DEAK calculus toolbox

The aim of the DEAK calculus tool [4] 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 can be found in the the folder `LaTeX` in [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,

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

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`

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

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

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

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

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

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. For example, with the shallow embedding, it will not be possible to write an Isabelle function that transforms a proof into a cut-free proof. (Cut-elimination is a topic we had to defer to future work, but we do make use of the deep embedding in the user interface.)

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:

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

takes a locale, a rule r, and a sequent s and outputs the list of premises needed to prove the s via r.[6] 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 it in future work to implement, and prove correct, cut elimination for D.EAK and related calculi.

3.3 Functionality of the user interface (UI)

For the reasons described at the end of Section 2, the UI is an essential part of the tool. The UI 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 (both SE and DE).
- 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 and atom 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.

[6] 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 [2].

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, is cross platform and allows the use of Java libraries. This was especially useful when creating the graphical interface 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.

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 [13] and by mathematical proofs about a formalisation in a Hilbert calculus [27]. It has also been automatically verified, for small values of k, using techniques from model checking [21] and automated theorem proving [12,28].

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, 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 after father's announcement.

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 [27, 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 ϕ' and f^k indicates k-fold integration 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 [27].

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 file `muddy-children.thy` [5].

The first part of `muddy-children.thy` 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 [27] 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 `muddy-children.thy` differs from its specification in [27] only in a few minor ways. Instead of assuming the axiom of introspection $[\mathtt{a}]p \to p$, 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 [27, Prop.24] in Isabelle. Part of this went into providing some 'infrastructure' contained in the files `NatToString.thy` and `DEAKDerivedRules.thy` 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 profited from 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 `muddy-children.thy`, 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. In accordance with the principle of modularity discussed in the introduction, 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 `muddy-children.thy` suffices.

5 Building your own calculus tool

As discussed in Section 3, the DEAK toolbox consists of a set of Isabelle theory files that formalize the terms and encode the 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.

On top of this, we provide the calculus toolbox, a meta-toolbox, which consists of a set of scripts and utilities used for maintaining and modifying the DEAK calculus tool and for building your own calculus tool.

The main component of the meta-toolbox is the build script, which takes in a description file of the terms and the rules of the calculus and expands this concise definition into multiple Isabelle theories and Scala code. Due to this centralised definition of the calculus, adding rules or logical connectives becomes much easier, as any changes made to the calculus 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 [3] and tutorial [2] is available.

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 calculus toolbox.

The calculus is described in a 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

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

`"type"` specifies that a sequent consists of two structures.[7] The next three lines specify how sequents will be typeset in Isabelle, ASCII and LaTeX. To make proofs readable in the UI, it is important that 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

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

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

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

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 (aka 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 exam-

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

ple 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

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

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 Γ, Δ are strings of action modalities. This is done by noting in the calculus description file the dependence on a condition called `atom` as follows.

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

The condition 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.

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

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 [17], 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.

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

To build the tool 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 tool-code is output to a directory called `gen_calc`.

Template files. The tool-code 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-BEGIN*)
auto-generated code ...
(*calc_structure-END*)
```

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 Conclusions

We find that the tool already makes a valuable contribution to our own, so far largely theoretical research. The main directions of future work consist in extending the tool to support more ambitious projects on the proof theory as well as on the application side. These may include the following concrete projects.

- Proving more theorems in and about D.EAK:
  - case studies similar to muddy children but with dynamic updates more complicated than public announcement,

  - proving cut elimination of D.EAK in Isabelle.
- Treating the multi-type version of D.EAK [16].
- Extending to other calculi. In particular, our methodology naturally applies to fragments of classical logics such as intuitionistic and linear logics which have many applications in computer science.
- Providing an interface for the calculus description file, possibly supporting the integration of different calculi.
- Making the tool more powerful by
  - extending available tactics,
  - improving the currently very rudimentary automated proof search.
- Prove bigger case studies of more substantial epistemic protocols. How will the tool need to change to scale it up to bigger applications?

Because in this paper we were interested in studying calculi such as D.EAK the propositional part about dynamic and epistemic operators and the higher order part of Isabelle are strictly separated, interfaced by the tool-generated shallow embedding. One should investigate building the dynamic and epistemic part directly into the higher-order logic of Isabelle. This would allow us to formalize properties such as "after m rounds of no each child knows that there are at least m dirty children" directly instead of encoding them as functions that map parameters such as m to formulas of propositional logic. On the other hand, one will loose information coming from the detailed understanding of a propositional, substructural, modular display calculus such as D.EAK.

Possibly the most important topic of further research concerns the fact that display calculi are not directly suitable for automatic proof search. On the other hand they have the advantages of modularity we discussed. So the question is whether we can—along the lines of [20]—go from display calculi constructed according to a clear proof theoretic methodology to deep inference calculi well suited for proof search. That automatic proof search is a direction worth pursuing for dynamic epistemic logics has been shown in [28]: for the calculus of [12] a depth first search augmented with simple heuristics was able to automatically find a proof of the muddy children puzzle for up to 4 dirty children, see [28, §6.4.3].

Another important problem concerns how to integrate common knowledge. This is a well-known difficult problem. Some proposed solutions use infinitary rules, other use finitary rules are non-standard and non-modular. We plan to extend D.EAK with common knowledge while keeping it modular.

References

1. F. Baader and C. Lutz. Description Logic. In *Handbook of Modal Logic*. 2006.
2. S. Balco. Building a sequent calculus toolbox. Available at http://goodlyrottenapple.me/2015/09/02/sequent-tutorial/.
3. S. Balco. The calculus toolbox. Download and documentation at https://github.com/goodlyrottenapple/calculus-toolbox.
4. S. Balco. The DEAK calculus tool. Download and documentation at https://github.com/goodlyrottenapple/DEAK-calculus-tool.
5. S. Balco and S. Frittella. Muddy children.thy. Isabelle 2015 theory file available at https://github.com/goodlyrottenapple/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 and J. van Benthem. Modal logic: A semantic perspective. In *Handbook of Modal Logic*. 2006.
9. P. Blackburn, J. van Benthem, and F. Wolter, editors. *Handbook of Modal Logic*. Elsevier, 2006.
10. 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.
11. J. E. Dawson and R. Goré. Formalised cut admissibility for display logic. In *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, pages 131–147, 2002.
12. 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.
13. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
14. M. Fitting. Modal Proof Theory. In *Handbook of Modal Logic*. 2006.
15. S. Frittella, G. Greco, A. Kurz, and A. Palmigiano. Multi-type display calculus for propositional dynamic logic. DOI:10.1093/logcom/exu064, 2014.
16. S. Frittella, G. Greco, A. Kurz, A. Palmigiano, and V. Sikimić. A multi-type display calculus for dynamic epistemic logic. DOI:10.1093/logcom/exu068, 2014.
17. S. Frittella, G. Greco, A. Kurz, A. Palmigiano, and V. Sikimić. A proof-theoretic semantic analysis of dynamic epistemic logic. *Journal of Logic and Computation*, 2015. DOI:10.1093/logcom/exu063.
18. R. Goré. On the completeness of classical modal display logic. *In H Wansing, editor, Proof Theory of Modal Logic*, volume 2 of Applied Logic:137–140, 1996.
19. R. Goré. Substructural logics on display. *Logic Journal of IGPL*, 6(3):451–504, 1998.
20. R. Goré, L. Postniece, and A. Tiu. Taming displayed tense logics using nested sequents with deep inference. In *Automated Reasoning with Analytic Tableaux and Related Methods, 18th International Conference, TABLEAUX 2009, Oslo, Norway, July 6-10, 2009. Proceedings*, pages 189–204, 2009.
21. 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). Cambridge, MA, USA, April 22-25, 1991.*, pages 325–334, 1991.
22. I. Horrocks, U. Hustadt, U. Sattler, and R. Schmitt. Computational Modal Logic. In *Handbook of Modal Logic*. 2006.
23. M. Kracht. Power and weakness of the modal display calculus. In *Proof theory of modal logic*, pages 93–121. Kluwer, 1996.

24. P. Lescanne. Mechanizing common knowledge logic using COQ. *Ann. Math. Artif. Intell.*, 48(1-2):15–43, 2006.
25. P. Lescanne. Common knowledge logic in a higher order proof assistant. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 271–284, 2013.
26. P. Lescanne and J. Puisségur. Dynamic logic of common knowledge in a proof assistant. *CoRR*, abs/0712.3146, 2007.
27. 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.
28. J. Truffaut. Implementation and improvements of a cut-free sequent calculus for dynamic epistemic logic, 2011. MSc thesis, University of Oxford.
29. H. P. van Ditmarsch and B. Kooi. *One Hundred Prisoners and a Light Bulb*. Springer, 2015.
30. H. P. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic Epistemic Logic*. Springer, 2007.
31. H. Wansing. *Displaying Modal Logic*. Kluwer, 1998.