Master Thesis

Implementing a Structured Approach to Belief Revision by Deterministic Switching Between Total Preorders

Heltweg, Philip
pheltweg@gmail.com

5th October 2021

Chair of Knowledge-Based Systems
Faculty of Mathematics and Computer Science
University of Hagen, Germany

Abstract

Belief change research investigates how agents adapt their knowledge with potentially conflicting information. A common formalization is by epistemic states, abstract entities often represented by faithful preorders. Operators describe how epistemic states change with new evidence and are classified by which postulates they satisfy. Different approaches have been suggested for the problem of iterated belief change. Recent work introduces uniform revision that revises an agent's beliefs based on one static total preorder, therefore lowering representational costs.

In this thesis, an extended epistemic state approach is introduced, based on an agent deterministically switching between total preorders. Challenges for implementations in the area of iterated belief change, like textual representation of total preorders, are pointed out and solutions developed. A tool for the automated certification of postulates for iterated belief change, called Coeus, is implemented for the new operator. Finally, the developed software is evaluated empirically. Coeus is publicly available, and most of its code is open-source.

I am thankful to Prof. Christoph Beierle and especially Kai Sauerwald for guiding me through the writing of this thesis as well as working on their team. Thanks to everyone at the chair of knowledge-based systems for their valuable feedback during this time.

Additionally, I would like to thank my parents, Mechthild and Hermann-Josef, for making me curious.

Finally, thank you, Caro, for your support, love, knowledge, help, patience... during this journey from start to finish. I love you.

Contents

1. Introduction

A fundamental skill for intelligent agents is updating their internal beliefs about the world with new, potentially conflicting information. The field of iterated belief change discusses a logic-based approach. The dominant framework considers an agent to possess an epistemic state, an abstract entity containing both its current belief set as well as a strategy for future belief revision when encountering new evidence. Based on the paradigm of AGM revision [AGM85], Darwiche and Pearl [DP97] have shown a model of iterated belief change as a plausibility ordering over possible worlds.

Because AGM revision only describes limited restrictions, iterated belief revision operators can be very complex. In fact, the amount of possible epistemic states can be infinite, leading to representational problems for artificial agents. Even if not infinite, as discussed by Areces and Becher [AB01], belief revision operators are relative to the original belief set of an agent and can be very complex. Many suggested operators rely on constructing new change strategies with every input.

Uniform revision [AB01, Ara20] proposes an approach to belief revision using a fixed structure that is not related to any current belief set of an agent. Because the change strategy itself does not change and can be reused, it provides an automatic solution to iterated belief revision.

The work investigates a class of revision operators based on the idea of deterministically switching between a finite set of preorders depending on input, introduced by Sauerwald and Beierle [SB21]. The operators are based on a directed graph, called a deterministic multiform system. This approach leads to an operator that is always representable in finite size. As every preorder encodes a revision strategy, and the associated deterministic multiform system encodes the change between them, this approach allows for an exploration of both. Since a directed graph underlies the representation of a deterministic multiform revision operator, methods of graph theory and finite state machines are applicable.

The goal of this work is to introduce the theoretical foundation of DMF revision operators and provide an implementation to support future research into DMF operators and their properties.

The thesis initially introduces uniform revision in the framework of plausibility orderings over worlds by Darwiche and Pearl. Based on uniform revision, deterministic multiform systems and DMF revision operators are defined.

A discussion of desirable properties for general research software is done as an in-

troduction to the implementation. Furthermore, challenges and possible solutions for implementations dealing with belief revision are presented. A subset of challenges, textual representation of formulas and tpos as well as performance, are described in detail.

Additionally, a tool is implemented that allows users to define DMF revision operators by describing deterministic multiform systems. The tool supports exporting and importing of DMF revision operators so they can be shared in future research. A limited set of belief revision postulates from literature is automatically checked by the tool using an extended WHIWAP [SH19] implementation.

To conclude, choices made during the implementation are evaluated, and alternative options are mentioned. A summary and further research opportunities provide an outlook.

2. Background

Belief change is the process of an intelligent agent to update its currently held beliefs with new information. It is often (but not necessarily) discussed in the context of propositional logic. In this chapter, prior work is introduced, starting with the seminal approach of AGM revision [AGM85] for belief sets. Furthermore, iterated belief revision and a common formalism in the form of epistemic states by Darwiche and Pearl [DP97] is described. The epistemic state framework provides a basis to discuss alternative approaches to iterated belief revision like extending epistemic states (e.g., Booth and Meyer [BM11]) or uniform revision [Ara20]. Uniform revision is introduced in the context of epistemic states and explained using an accompanying example as it will be the basis for deterministic multiform revision discussed in Section 3.

2.1. Notation

Let $\mathcal{L}$ be a propositional language over a finite signature Σ. Lower case letters $a, b, c...$ are used to denote propositional variables, lower greek letters α, β, γ for formulas in $\mathcal{L}$ and $\top, \bot$ represent tautology and contradiction respectively. For a set X, $\mathcal{P}(X)$ is it's power set. $\equiv$ classical logical equivalence.

The set of propositional interpretations is denoted by Ω, also referred to as (possible) worlds. A world $\omega \in \Omega$ can be expressed as a sequence of all propositional variables. Overlining is used to show an assignment of false in the world ω. For example $\overline{ab}c$ stands for a world that assigns a, b to $false$ and c to $true$.

$\models$ stands for propositional entailment. Given a formula $\alpha \in \mathcal{L}$, the set of worlds that satisfy α is denoted by $[\![\alpha]\!]$. $th(W)$ stands for the set of sentences true in all worlds $W \in \Omega$. $Cn(X) = \{\beta | X \models \beta\}$ is the deductive closure of a set of formulas X. For ease of notation, the deductive closure of individual formula $Cn(\{\alpha\})$ is written as $Cn(\alpha)$. The set of all belief sets (deductively closed sets of formulas over $\mathcal{L}$) is denoted as $\mathcal{B} = \{F | F \in \mathcal{P}(\mathcal{L}) \text{ and } Cn(F) = F\}$.

A total preorder (tpo for short) $\leq \in \Omega \times \Omega$ is a total, reflexive and transitive relation. The strict part of $\leq$ is denoted as $<$ while $\sim$ stands for its symmetric closure.

2.2. AGM Theory

AGM theory (Alchourrón, Gärdenfors, and Makinson in [AGM85]) deals with belief change in belief sets. A belief set $B \in \mathcal{B}$ is the deductively closed set of propositions an

agent accepts as true at any given point in time [FH11]. Belief change can be categorized into expansion, contraction, or revision with new information. Expansion is the process of incorporating a new piece of information that is not inconsistent with currently held beliefs. Contraction refers to giving up a belief that has become questionable. The process of belief revision refers to keeping an agent's set of beliefs consistent while incorporating new information that can be inconsistent with the current belief set [KM91, Gä84].

The AGM approach to belief change is defined by a binary function (also called operators) $*$ that transitions a consistent belief set B and an input sentence α to a consistent belief set $B * \alpha$. A set of postulates for what is considered rational revision of belief sets is introduced in [AGM85] and presented here in the formulation of Aravanis [Ara20]. $B * \alpha$ denotes the revision of a belief set B by α while $B + \alpha$ denotes the expansion.

(AGM$*$1) $B * \alpha$ is a belief set

(AGM$*$2) $\alpha \in B * \alpha$

(AGM$*$3) $B * \alpha \subseteq B + \alpha$

(AGM$*$4) If $\neg\alpha \notin B$, then $B + \alpha \subseteq B * \alpha$

(AGM$*$5) $B * \alpha \models \bot$ iff $\models \neg\alpha$

(AGM$*$6) If $Cn(\alpha) = Cn(\beta)$, then $B * \alpha = B * \beta$

(AGM$*$7) $B * (\alpha \wedge \beta) \subseteq (B * \alpha) + \beta$

(AGM$*$8) If $\neg\beta \notin B * \alpha$, then $(B * \alpha) + \beta \subseteq B * (\alpha \wedge \beta)$

Definition 1 (AGM revision operator for belief sets)**.** *Functions* $* : \mathcal{B} \times \mathcal{L} \to \mathcal{B}$ *that map belief sets and an input formula to a new belief set are called AGM revision operators for belief sets if they satisfy the postulates (AGM*$*$*1) - (AGM*$*$*8).*

2.3. The Epistemic State Framework

While the original AGM postulates describe a family of functions for belief set revision, they do not define concrete operators. A constructive approach was presented by Darwiche and Pearl in [DP97]. They distinguish between beliefs an agent accepts and are

part of the belief set and conditional beliefs, beliefs an agent is prepared to adopt with new evidence. While two agents might have the same current beliefs (i.e., the same belief set), they might still come to different conclusions with new information depending on their previous worldview. Darwiche and Pearl introduced epistemic states, abstract entities that include all information an agent needs to encode not only its currently held beliefs but also a strategy for belief change with future input. This work will refer to the set of all epistemic states over $\mathcal{L}$ as $\mathcal{E}$.

Definition 2 (Belief revision operator in the epistemic state framework)**.** *In the epistemic state framework a belief revision operator is a function* $\circ : \mathcal{E} \times \mathcal{L} \to \mathcal{E}$.

Since epistemic states are abstract entities, they can only be reasoned over with the help of a representation theorem that assigns them to a concrete structure. In the faithful-preorders model introduced by Katsuno and Mendelzon [KM91] and extended to epistemic states by Darwiche and Pearl [DP97], epistemic states can be equipped with total preorders over Ω. For this work, it is assumed that a consistent set of propositions $Bel(\Psi)$ can be extracted from every epistemic state $\Psi \in \mathcal{E}$. The models of an epistemic state are defined as the models of its associated belief set, $[\![\Psi]\!] = [\![Bel(\Psi)]\!]$. So-called faithfulness ensures the compatibility to $Bel(\Psi)$.

Definition 3 (Faithful assignment [DP97])**.** *A function* $\Psi \mapsto \leq_\Psi$ *that maps each epistemic state to a total preorder on interpretations is said to be a faithful assignment if:*

(FA1) *If* $\omega_1, \omega_2 \in [\![\Psi]\!]$*, then* $\omega_1 \sim_\Psi \omega_2$

(FA2) *If* $\omega_1 \in [\![\Psi]\!]$ *and* $\omega_2 \notin [\![\Psi]\!]$*, then* $\omega_1 <_\Psi \omega_2$

Intuitively, such a faithful-tpo orders possible worlds according to their assumed plausibility, with lower-ranked worlds being more plausible than higher-ranked ones. A faithful assignment does not uniquely assign a preorder to a given epistemic state but describes the structure of a family of preorders assigned to the epistemic state.

In this model, belief revision of belief sets can be expressed as follows.

Definition 4 (Constructing an AGM revision operator from a tpo)**.** *For notation, let W be a set of worlds with $W \subseteq \Omega$ and $min(W, \leq_\Psi)$ denote the set of minimal worlds in W under $\leq_\Psi$. An AGM revision operator for belief sets (Definition 1) can be constructed from this family of preorders with the following condition (as shown by Katsuno and Mendelzon [KM91]).*

$$B * \alpha = th(min([\![\alpha]\!], \leq_\Psi))$$

Using the condition from Definition 4 and the definition of a faithful assignment (Definition 3) is the basis for AGM revision operators on epistemic states.

Definition 5 (AGM revision operator for epistemic states [DP97])**.** *A belief revision operator $\circ$ is an AGM revision operator for epistemic states if there is a faithful assignment $\Psi \mapsto \leq_\Psi$ such that:*

$$[\![\Psi \circ \alpha]\!] = min([\![\alpha]\!], \leq_\Psi)$$

In this work, all operators are assumed to fulfil the following syntax-independent condition:

$$\text{if } \alpha \equiv \beta, \text{ then } \Psi \circ \alpha = \Psi \circ \beta$$

2.4. Iterated Belief Revision

A common requirement for belief revision operators is to be able to handle consecutive inputs, so-called iterated belief revision. For this purpose, an iterated belief revision operator on epistemic states $\circ$ transitions an epistemic state Ψ with new information to $\Psi \circ \alpha$. In the next step that state becomes the input for an iteration step by information β: $(\Psi \circ \alpha) \circ \beta$. Since the AGM postulates only concern revision operator for belief sets, not epistemic states encoding the strategy an agent is employing to change their beliefs, they place no restrictions on how these conditional beliefs change.

The following section presents a selection of previous approaches to iterated belief revision that are relevant to the work, a more general overview can be found from Peppas [Pep13].

In the faithful-tpo model, the iterated change of epistemic states can be described by the change in the family of assigned tpos. Darwiche and Pearl define postulates that restrict how the orderings (and therefore the epistemic state) undergo change. Their semantic version is presented here.

(CR1) if $\omega_1, \omega_2 \in [\![\alpha]\!]$, then $\omega_1 \leq_\Psi \omega_2 \leftrightarrow \omega_1 \leq_{\Psi \circ \alpha} \omega_2$

(CR2) if $\omega_1, \omega_2 \in [\![\neg\alpha]\!]$, then $\omega_1 \leq_\Psi \omega_2 \leftrightarrow \omega_1 \leq_{\Psi \circ \alpha} \omega_2$

(CR3) if $\omega_1 \in [\![\alpha]\!]$ and $\omega_2 \in [\![\neg\alpha]\!]$, then $\omega_1 <_\Psi \omega_2 \rightarrow \omega_1 <_{\Psi \circ \alpha} \omega_2$

(CR4) if $\omega_1 \in [\![\alpha]\!]$ and $\omega_2 \in [\![\neg\alpha]\!]$, then $\omega_1 \leq_\Psi \omega_2 \rightarrow \omega_1 \leq_{\Psi \circ \alpha} \omega_2$

(CR5) if $\omega_1, \omega_3 \in [\![\alpha]\!]$ and $\omega_2 \in [\![\neg\alpha]\!]$, then $\omega_3 <_\Psi \omega_1$ and $\omega_2 \leq_\Psi \omega_1 \rightarrow \omega_2 \leq_{\Psi \circ \alpha} \omega_1$

(CR6) if $\omega_1, \omega_3 \in [\![\alpha]\!]$ and $\omega_2 \in [\![\neg\alpha]\!]$, then $\omega_3 <_\Psi \omega_1$ and $\omega_2 <_\Psi \omega_1 \rightarrow \omega_2 <_{\Psi \circ \alpha} \omega_1$

Another possible solution is to enrich the epistemic state with additional structures to revise a tpo. One such approach, that revises a tpo by employing an additional interval based ordering, is described by Booth and Meyer in [BM11]. It is important to note that, for the enriched epistemic states approach, the problem of iteration also has to be solved for the new structure.

2.5. Uniform Revision

Most approaches discussed in belief change share the assumption of idealized agents. In a purely theoretical context, it does not matter how complex a belief change operator is. However, practical applications have to deal with the limited capabilities of real-world agents like humans or restricted hardware. In these cases, a belief revision operator does not only need to work theoretically but must also be computable in a reasonable time.

In their work on iterated AGM functions, Areces and Becher [AB01] point out that AGM revision operators are relative to a belief set. Therefore, for iterated belief revision, the revision operator also needs to be revised with every input, increasing complexity. Instead, they propose true binary functions that directly map any belief set and input to a new belief set but not change themselves. These functions can be used for iteration, as the new belief set is a valid input for the same function.

Building on iterated AGM functions, Aravanis [Ara20] discusses uniform revision. Uniform revision operators encode the change strategy of an agent in a single, fixed total preorder over all worlds. In this view, the fixed tpo describes rules of the domain that never change, e.g., laws of physics.

Restricting the structure to only one total preorder has considerable benefits. In contrast to other revision operators, the representational costs for uniform revision are

low as a single tpo completely defines an operator. As a result, uniform revision operators are simpler to understand and define than approaches that encode a tpo for every possible epistemic state. Lastly, because the tpo does not change with input, uniform revision can be used for iterated revision without additional overhead: The same operator is defined for every possible belief set of an agent and can be reused.

Aravanis gives a constructive approach for uniform revision operators by considering a total preorder $\preceq$ over all worlds that is fixed and independent of the state of an agent. It encodes the domain knowledge of the agent that is separated from its concrete belief set.

Example 1. As an accompanying example, consider an agent deciding if they can buy bread or not. Let a represent "Bakery A is open", and b represent "Bakery B is open".

Normally, one or both bakeries will be open. A fixed tpo in uniform revision $\preceq$, representing the agents assumptions could be $ab \sim a\overline{b} \sim \overline{a}b \preceq \overline{ab}$.

Booth and Meyer [BM06] introduce a visualization of tpos as a linearly ordered set of ranks. The ranks of worlds in a tpo $\leq$ are defined as the equivalence classes modulo the symmetric closure of $\leq$: $[[x]]_\sim = \{y \mid y \sim x\}$ and then ordered by the relation $[[x]] \leq [[y]]$ iff $x \leq y$. A similar visualization will be used in this work, Figure 1 shows $\preceq$ with the ranks ordered by plausibility from bottom to top and their index on the left. The most plausible worlds are shown in the lowest rank with the index 0.

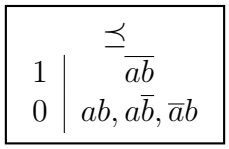


Figure 1: A tpo for Example 1, visualized as an linearly ordered set of ranks, lower ranks are more plausible.

.

Let $\Psi, \Phi \in \mathcal{E}$ be epistemic states. In the epistemic state framework, a uniform revision operator is induced from $\preceq$ by a faithful-tpo $\leq_\Psi$ that satisfies one of the following conditions.

(URF1) For any $\omega_1, \omega_2 \notin [\![\Psi]\!]$, $\omega_1 \leq_\Psi \omega_2$ iff $\omega_1 \preceq \omega_2$

(URF2) For any $\omega_1, \omega_2 \notin [\![\Psi]\!] \cup [\![\Phi]\!]$, $\omega_1 \leq_\Psi \omega_2$ iff $\omega_1 \leq_\Phi \omega_2$

The condition (URF1) means that any worlds not in the belief set of the agent are ordered according to $\preceq$. A condition that does not require $\preceq$ is (URF2). Any worlds that are not in both belief sets are ordered equivalently for any two epistemic states. (URF1) holds for a tpo iff (URF2) holds ([Ara20]).

These conditions define a set of preorders, one for every belief set $B \in \mathcal{B}$. From this, an AGM revision operator is defined by Definition 4. An operator constructed in this manner is referred to as uniform revision operator $*_{UR}$.

Example 2. Let the agent be in a state Ψ assuming both bakeries are open: $[\![\Psi]\!] = \{ab\}$. A tpo $\leq_\Psi$, faithful to $[\![\Psi]\!]$ and satisfying (URF1) for the fixed tpo $\preceq$ from Example 1, is shown in Figure 2. Notice how the world ab is in the most plausible rank (due to the faithfulness from Definition 3). All worlds that are not in $[\![\Psi]\!]$ are ordered according to the fixed tpo $\preceq$. The tpo $\leq_\Psi$ defines a uniform revision operator, in relaxed notation $*$, as described in Definition 4.

	$\leq_\Psi$
2	$\overline{a}\overline{b}$
1	$\overline{a}b, a\overline{b}$
0	ab

Figure 2: A tpo $\leq_\Psi$, faithful to $[\![\Psi]\!] = \{ab\}$

.

The agent now learns a local festival is taking place. During some festivals, either essential businesses like bakeries have to close, or both will be open due to the increased demand. An input modeling this could be $\alpha = a \leftrightarrow b$. Because this information is not contradictory to the current belief set of the agent, $[\![Bel(\Psi) * \alpha]\!] = \{ab\}$.

	$\leq_{\Psi * \alpha}$
2	$\overline{a}\overline{b}$
1	$\overline{a}b, a\overline{b}$
0	ab

Figure 3: The tpo from Figure 2 stays the same after revision by $\alpha = a \leftrightarrow b$.

.

In a second step, the agent learns that Bakery A is closed, $\beta = \neg a$. This information forces the agent to revise its belief set because $\neg a$ is contradictory to its currently held belief of $a \wedge b$. With the same construction, the uniform revision is $[\![Bel(\Psi) * \alpha * \beta]\!] = \{\overline{a}b\}$. The faithful tpo $\leq_{\Psi * \alpha * \beta}$ that satisfies (URF1) as well is shown in Figure 4. In this case,

revision by $\neg a$ causes the agent to ignore the previous information and conclude that b, Bakery B is open. The new belief makes sense under the laws for normal days but fails to take into account special holidays.

$\leq_{\Psi * \alpha * \beta}$	
2	$a\overline{b}$
1	$ab, a\overline{b}$
0	$\overline{a}b$

Figure 4: The tpo for $\Psi * \alpha * \beta$ after uniform revision by $\beta = \neg a$.

.

Notice that the revision ignores historical information because a uniform revision operator is always defined relative to $\preceq$ and the current belief set of an agent only. As long as the current belief set is the same, any previous history does not influence the revision.

3. Deterministic Multiform Revision

As introduced in Subsection 2.5, uniform revision is a simple and efficient form of belief revision. However, it is also more limited than other approaches. Multiform revision aims to build on uniform revision to make it more flexible but keep the main advantages of low representational costs and intuitive definition. The core idea of deterministic multiform revision is to extend uniform revision with concepts of finite state machines of computer science [SB21].

Conceptually, multiform revision encodes a finite set of contexts (in the form of tpos $\preceq$ as in uniform revision) and rules for switching between them in a directed graph, called a multiform system. While agents change their contexts with new information, the underlying system does not need to be revised. Because the tpos in the multiform system induce different uniform revision operators, an agent is essentially changing between a finite set of uniform revision operators with new information.

3.1. Multiform Systems

Multiform revision is an approach to iterated belief revision build on the idea of enriching the epistemic state (as discussed in Subsection 2.4). A multiform system describes a finite set of contexts and rules for switching between them that an agent can utilize for its revision strategy.

Definition 6 (Multiform System)**.** *A tuple $M = (T, E)$ is called a multiform system (for $\mathcal{L}$) if T is a finite set of total preorders over Ω and $E \subseteq T \times \mathcal{L} \times T$.*

A multiform system can be thought of as a directed graph with the nodes being total preorders and edges labeled with formulas. Figure 5 shows an example of a multiform system.

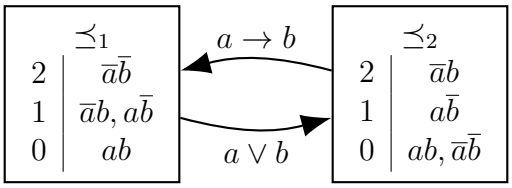


Figure 5: Example of a multiform system as a graph

This work assumes the additional restriction of only deterministic change between tpos. A multiform system that allows only a single context transition with new inform-

ation is called deterministic and abbreviated as a DMF system. DMF systems discussed in this work are assumed to be deterministic and syntax-independent multiform systems.

For ease of notation, let $E(\preceq, \alpha)$ refer to the set of total preorders that is directly reachable from $\preceq$ via the α edge: $\{\preceq' \mid (\preceq, \alpha, \preceq') \in E\}$. Additionally, $E(\preceq, \alpha) = \preceq'$ is used to refer to $E(\preceq, \alpha) = \{\preceq'\}$.

Definition 7 (Deterministic Multiform System)**.** *A multiform system $M = (T, E)$ is called a deterministic multiform system if $|E(\preceq, \alpha)| = 1$. A multiform system is called* syntax independent *if $\alpha \equiv \beta$ implies $(\preceq, \alpha, \preceq') \in E$ iff $(\preceq, \beta, \preceq') \in E$.*

Similar to the tpo $\preceq$ in uniform revision, DMF systems do not change with new information. Instead, the enriched epistemic state of an agent contains enough information to construct a belief revision operator for new input and transition to a new context.

3.2. Deterministic Multiform Revision

In the epistemic state framework, an epistemic state $\Psi \in \mathcal{E}$ is an abstract entity encoding the currently held beliefs and change strategy of an agent. A possible instantiation of these abstract entities is a tuple of belief set and a tpo from an underlying DMF system called a DMF system state.

Definition 8 (DMF system state)**.** *Let $M = (T, E)$ be a* deterministic multiform system*. A tuple $(\mathbb{B}, \preceq)$ is referred to as a* DMF system state *of M, denoted as s^M if $\mathbb{B} \in \mathcal{B}$ is a belief set and $\preceq \in T$ is a tpo over Ω.*

A DMF system state encapsulates the beliefs of an agent. It defines its *currently accepted beliefs* in the form of a belief set $\mathbb{B}$. In addition, it defines the *context* (what Aravanis calls "state-independent dynamics of a particular domain" [Ara20]) for its decision making by referencing a tpo $\preceq \in T$.

In contrast to uniform revision, a DMF system state (with the underlying DMF system M) allows an agent to change between a finite set of prepared contexts T with new information. These context switches could, e.g., represent different legal frameworks like switching from common law in the USA to civil law in Germany. Similarly, different contexts could play a role in medical diagnosis. For example, new laboratory results that could indicate pregnancy would lead to different assumptions if the context is a male patient. For a topical example at the time of writing, an agent would be more likely to diagnose the common flu instead of COVID-19 if the context is a vaccinated patient.

Compatibility with the belief set of an abstract epistemic state Ψ with a concrete DMF system state over M, s^M_Ψ is ensured by a DMF system state assignment.

Definition 9 (DMF system state assignment)**.** *Let M be a deterministic multiform system. A function $\Psi \mapsto s^M_\Psi$, mapping each epistemic state to a* DMF system state *$s^M_\Psi = (\mathbb{B}_\Psi, \preceq_\Psi)$ of M is called a* DMF system state assignment *over M if $Bel(\Psi) = \mathbb{B}_\Psi$.*

The information encoded in a DMF system state induces a uniform revision operator to revise belief sets. The context tpo $\preceq \in T$ of the DMF system state is comparable to the unique, fixed tpo in uniform revision. It defines a set of tpos (one for every belief set), as described in Subsection 2.5 from which a tpo is selected for being faithful to $Bel(\Psi)$.

Definition 10 (Revision operator induced by a DMF system state)**.** *Let s^M be a* DMF system state *and $\preceq \in T$ its context tpo. For every belief set $B \in \mathcal{B}$ a unique tpo $\leq_B$ is induced by:*

- *Satisfying condition (URF1) / (URF2) in regards to $\preceq$*
- *Being faithful in regards to B*

The tpo $\leq_B$ defines an AGM revision operator for belief sets as per Definition 4:

$$B * \alpha = th(min([\![\alpha]\!], \leq_B))$$

This revision operator will be referred to as $*_{s^M_\Psi} : \mathcal{B} \times \mathcal{L} \to \mathcal{B}$ and is defined for any DMF system state *s^M_Ψ. It is a uniform revision operator in regards to $\preceq$, as defined by Aravanis [Ara20].*

The revision operator induced by a DMF system state $*_{s^M}$ plays a central role in the revision of epistemic states based on a DMF system M. Intuitively, an agent employs uniform revision (in the current context $\preceq$) to revise its belief set. Expanding on uniform revision, the agent also optionally switches the overall context, depending on the underlying DMF system.

More formally, let $M = (T, E)$ be a *deterministic multiform system* and s^M_Ψ be the *DMF system state* assigned to an epistemic state Ψ by a *DMF system state assignment*. The uniform revision operator induced by s^M_Ψ is denoted as $*_{s^M_\Psi}$. Then a DMF system revision operator for epistemic states is defined in the following manner.

Definition 11 (DMF system revision operator for epistemic states)**.** *A revision operator for epistemic states $\circ : \mathcal{E} \times \mathcal{L} \to \mathcal{E}$ is called a* DMF system revision operator for epistemic

states *if there is a* DMF system state assignment $\Psi \mapsto s_\Psi^M$ *over* M *such that* $s_{\Psi\circ\alpha}^M = (\mathbb{B}_{\Psi\circ\alpha}, \preceq_{\Psi\circ\alpha})$ *for every* Ψ *and* α *as follows:*

$$\mathbb{B}_{\Psi\circ\alpha} = \mathbb{B}_\Psi *_{s_\Psi^M} \alpha$$

and

$$\preceq_{\Psi\circ\alpha} = \begin{cases} \preceq' & \text{if } (\preceq_\Psi, \alpha, \preceq') \in E \\ \preceq_\Psi & \text{otherwise} \end{cases}$$

A problem that arises in similar approaches to iterated revision with enriched epistemic states (e.g., for Booth and Meyer [BM11]) is the need for an additional revision strategy of the new structure. An advantage to revision based on DMF systems is that the underlying structure can stay static and does not need to be revised with new information. Definition 11 is therefore sufficient to define an approach to iterated revision.

Definition 12 (DMF system revision operator for belief sets)**.** *Let* $\circ$ *be an* DMF system revision operator for epistemic states *according to Definition 11 for a* DMF system M*. A revision operator for belief sets* $* : \mathcal{B} \times \mathcal{L} \to \mathcal{B}$ *is called a* DMF system revision operator for belief sets *if there is a* DMF system state assignment $\Psi \mapsto s_\Psi^M$ *over* M *with* $s_\Psi^M = (\mathbb{B}_\Psi, \preceq_\Psi)$ *such that* $Bel(\Psi) * \alpha = \mathbb{B}_{\Psi\circ\alpha}$ *for every* Ψ *and* α*.*

Intuitively, an agent employing DMF revision extends uniform revision by switching between a finite set of contexts T (and therefore a finite set of uniform revision operators) depending on new information. Reasons for those context switches are encoded in the edges E of the underlying DMF system M. As long as new information does not lead to a context switch, the agent revises their belief set using the same uniform revision operator. After receiving information that leads to a context switch, the agent revises their belief set and changes to a new uniform revision operator.

3.3. Postulates and DMF Revision

In literature, belief revision operators are grouped into classes that are defined by a set of postulates. Well known examples are the previously stated AGM postulates (AGM$*$1) - (AGM$*$8) [AGM85] for the class of *rational revision* operators or the Darwiche and Pearl postulates (CR1) - (CR6) [DP97]. Following the introduction of *DMF revision operators*, the following sections will discuss how to approach their classification.

The approach is twofold. First, a representation of a DMF system state as a derived-faithful tpo is introduced. Second, a browser-based software, called Coeus, is presented that allows the automated verification of operators for a belief change step, extending a previous implementation called WHIWAP by Sauerwald und Haldimann [SH19]. The implementation allows for empirical research into belief change based on *DMF revision operators* and can be used in further research or didactic purposes.

Postulates in literature are commonly given either as syntactic postulates or semantic postulates. Syntactic postulates describe changes of belief sets, while semantic postulates describe changes in a semantic domain, e.g., the common faithful preorder formalization. Often, representation theorems exist to express syntactic postulates in the semantic domain.

WHIWAP (and by extension Coeus) verifies belief change postulates by checking the difference between epistemic states represented by tpos. A derived-faithful tpo to a DMF system state is introduced here to verify postulates in the semantic domain of faithful preorders for DMF revision steps.

Definition 13 (Derived-faithful tpo to a DMF system state)**.** *A tpo* $\leq_{s_\Psi^M}$ *is called derived-faithful to a* DMF system state $s_\Psi^M = (\mathbb{B}_\Psi, \preceq_\Psi)$ *iff:*

1. *It is* faithful *to the belief set* $\mathbb{B}_\Psi$ *of* s_Ψ^M*:* $\mathbb{B}_\Psi = th(min([\![\top]\!], \leq_{s_\Psi^M}))$
2. *Other conditional beliefs encoded by it align with the state context* $\preceq_\Psi$*: For any* $\omega_1, \omega_2 \notin [\![\mathbb{B}_\Psi]\!]$*,* $\omega_1 \leq_{s_\Psi^M} \omega_2$ *iff* $\omega_1 \preceq_\Psi \omega_2$

These conditions correspond to the constructive approach used in Definition 10. Condition 2 is equivalent to (URF1) by Aravanis [Ara20], inducing a family of tpos for $\preceq_\Psi$. A unique tpo $\leq_{s_\Psi^M}$ is selected from that family by condition 1 and the belief set $\mathbb{B}_\Psi$ of s_Ψ^M.

Because the derived-faithful tpo fulfills the faithfulness condition, it encodes the same currently held beliefs as the belief set $\mathbb{B}_\Psi$ of the agent in the DMF system state instantiation.

The AGM revision operator constructed from a derived-faithful tpo according to Definition 4, represents the same revision strategy for belief sets encoded in the DMF system state itself. Condition 2 ensures that the belief revision operator induced by the derived-faithful tpo is equal to the revision operator induced by the DMF system state according to Definition 10. The conditionally held beliefs, in the form of a plausibility ordering over worlds, also align to the context tpo from the DMF system state.

For a single belief change based on DMF revision with a DMF system M, Coeus verifies postulates for the derived-faithful tpos of the prior and posterior epistemic state. The structure of M restricts how the derived-faithful tpos can change and therefore influences what postulates are fulfilled with each change.

Example 3. Recall the situation introduced in Example 1 and discussed for uniform revision in Example 2. While relying on uniform revision, the agent encoded the unchanging rules of the environment in one fixed tpo $\preceq$. Because the opening of local bakeries is managed differently depending on holidays, this could lead to wrong conclusions.

In DMF revision, these different contexts can be modelled. A minimal DMF system $M = (T, E)$ with two contexts $T = \{\preceq_1, \preceq_2\}$ and edges $E = \{(\preceq_1, a \leftrightarrow b, \preceq_2), (\preceq_2, a \wedge b, \preceq_1)\}$ is shown in Figure 6. The tpo $\preceq_1$ is equal to $\preceq$ of Example 2, encoding the laws for normal days. However, the agent switches its context to $\preceq_2$, encoding laws for a holiday, when receiving the information $a \leftrightarrow b$.

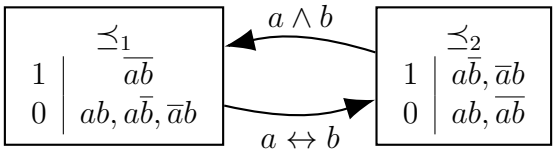


Figure 6: A DMF system for Example 3

An initial DMF system state that represents the agent assuming that both bakeries are open on a normal day is $s_\Psi^M = (Cn(a \wedge b), \preceq_1)$. Figure 7 shows its derived-faithful tpo.

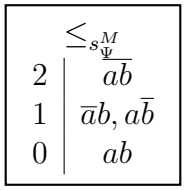


Figure 7: The derived-faithful tpo $\leq_{s_\Psi^M}$ to the DMF system state $s_\Psi^M = (Cn(a \wedge b), \preceq_1)$
.

On learning that a holiday means either both or no bakeries will be closed ($\alpha = a \leftrightarrow b$), the agent revises its epistemic state according to Definition 11. As in Example 2, because the input does not contradict the current beliefs, its belief set stays the same. In contrast to uniform revision however, the agent does switch its context tpo from $\preceq_1$ to $\preceq_2$ over the edge $(\preceq_1, a \leftrightarrow b, \preceq_2)$.

The new DMF system state is $s^M_\Phi = s^M_{\Psi\circ\alpha} = (Cn(a \wedge b), \preceq_2)$ and its derived-faithful tpo is displayed in Figure 8. Notice how the agents conditional beliefs, the beliefs it is prepared to accept with new information, have now changed to the new context.

In this way, DMF revision allows an agent to take historical information into account for future revision. Any information that does not substantially change the context can still be ignored, like in uniform revision. However, if an input changes the fixed rules of the domain (in this case, laws for the opening of businesses), the agent will also change its revision strategy accordingly.

	$\leq_{s^M_\Phi}$
2	$\overline{a}\overline{b}, a\overline{b}$
1	$\overline{ab}$
0	ab

Figure 8: The derived-faithful tpo $\leq_{s^M_\Phi}$ to the DMF system state $s^M_\Phi = (Cn(a \wedge b), \preceq_2)$

.

In this example, when learning Bakery A is closed ($\beta = \neg a$), the agent comes to a different conclusion than in Example 2. Because of the change in context tpo from $\preceq_1$ to $\preceq_2$ the minimal model of β is now $min([\![\beta]\!], \leq_{s^M_\Phi}) = \{\overline{ab}\}$. Accordingly the agents new DMF system state will be $s^M_\Xi = s^M_{\Phi\circ\beta} = (Cn(\neg a \wedge \neg b), \preceq_2)$ and it concludes $\neg b$, Bakery B is closed as well. The corresponding derived-faithful tpo $\leq_{s^M_\Xi}$ is shown in Figure 9.

	$\leq_{s^M_\Xi}$
2	$\overline{a}\overline{b}, a\overline{b}$
1	ab
0	$\overline{ab}$

Figure 9: The derived-faithful tpo $\leq_{s^M_\Xi}$ to the DMF system state $s^M_\Xi = (Cn(\neg a \wedge \neg b), \preceq_2)$

.

4. Towards Automated Certification of Postulates

Parts of this section were already published in joint work with Christoph Beierle and Kai Sauerwald [SH21, SHB21].

Iterated belief change often is frequently modeled by operators over epistemic states. These postulates are axiomatically classified by which postulates they fulfill, e.g., the Darwiche and Pearl postulates for iterated belief revision introduced in Subsection 2.4.

Therefore, a common problem is checking if a given postulate is satisfied by a belief change or a whole operator, more formally: Given a belief change operator $\circ$ and a postulate P, does $\circ$ satisfy P? In this work, the problem will be called the *Certification Problem*. Several sub-problems were identified to approach the automation of the certification problem:

1. A singular belief change from Ψ to Ψ by α
2. Iterated belief change with consecutive sentences α, β, ...
3. All singular belief changes on a state Ψ

The following sections identify challenges and possible solutions to automate the certification of postulates. A concrete implementation for iterated belief revision based on DMF revision is presented in Section 5.

4.1. Goals

A previous implementation for certification of a singular belief change was available with WHIWAP [SH19]. This work aims to extend WHIWAP in multiple ways. First by carefully formalizing the approach using model-checking in cooperation with Christoph Beierle and Kai Sauerwald [SH21, SHB21]. Second, by developing a tool for the automated certification of postulates in DMF revision that is highly modularized and can be extended further. Finally, by allowing users to submit multiple inputs, therefore verifying postulates for consecutive belief changes.

The implementation can support researchers with experimental studies of belief revision. Additionally, it has applications in an education environment to show a concrete approach to iterated belief revision. As a guideline during development, the following goals were identified:

1. Simple distribution and setup to make it accessible to a wider audience

2. Allow configuration of more than one belief change
3. Visualizations and automated computation of the next epistemic states for increased ease of use
4. Configuration should be shareable, both to support researchers in distributing their findings as well as to provide examples for students
5. Extensibility with new, potentially user-defined postulates to aid experimental studies

In addition, as a software artifact, it should also fulfill a baseline of properties independent of belief revision itself. The FAIR principles were considered as starting point. Originally defined for data by Wilkinson et al. [WDA+16] they provide a guideline for managing research artifacts to increase future impact. They have been expanded to research software by Hasselbring et al. [HCH+20]. The principles are:

1. Findable: Be easy to find and cite
2. Accessible: Provide a way to download a software snapshot
3. Interoperable: Use existing standards and target as many runtime platforms as possible
4. Reusable: Modular and easy to extend, follow good software development practices

In the context of research software, the FAIR principles increase reproducibility and reusability. Because the tool should be useful for experimental studies, it must be easy to configure and share by other researchers as well. The exported configuration should be in an open and standard format. Additionally, the tool will likely be extended with future work by students. Making the software modular and following good practices like automated testing will make their contribution easier.

Finally, to support the concrete application of belief revision by DMF revision, the tool should:

1. Support the configuration of a DMF system and initial DMF system state
2. Allow a user to create a belief revision input and definition of a subsequent DMF system state
3. Compute derived-faithful tpos for the prior and posterior DMF system state and verify if they satisfy a set of postulates or not

4.2. Encoding as Model-Checking

By employing model checking, the certification of a belief change satisfying a postulate or not can be completely automated. In addition, the approach extends naturally to new postulates. The same algorithm can be reused to certify belief changes by providing a postulate as a first-order formula. It would also be possible to provide counterexamples for postulates that are not satisfied by a belief change. The main challenge to employ model-checking is the scaling with input size, meaning considerable work has to be done for performance optimization.

During the writing of this thesis, the approach was described in more detail by Christoph Beierle, Kai Sauerwald, and the author in a paper. The results were submitted and accepted at the International Workshop on Nonmonotonic Reasoning 2021 [SHB21], an extended version was accepted at the Workshop on Formal and Cognitive Reasoning 2021 [SH21]. A summary is provided here.

First, a first-order logic fragment FO^{TPC} (for "Total Preorder Change") is defined to encode changes in epistemic states with new information. Based on this language, a FO^{TPC}-structure A_C is created for a belief change step $C = (\Psi_0, \alpha, \Psi_1)$ from a prior epistemic state Ψ_0 to a posterior epistemic state Ψ_1 by input α. Postulates are represented as formulas over FO^{TPC}. A postulate, represented by a formula φ is then satisfied by the belief change C iff $A_C \models \varphi$.

An initial study of belief change literature [DP97, BM06, JT07, Boo02, NPP03] was done to define a minimal set of predicates and functions commonly found in postulates. An overview is shown in Figure 10, examples include predicates such as $Mod(w, x)$ (w is a model of x), $Int(w)$ (w is an interpretation) or functions like $op(e_0, a)$ (the result of changing e_0 by a). e_0 and a are reserved as constants to refer to the prior epistemic state and the input formula, respectively.

From this minimal set, additional predicates can be constructed as needed, e.g., logical implication:

$$LogImpl(x, y) \coloneqq \forall w. Int(w) \to Mod(w, x) \to Mod(w, y)$$

A finite FO^{TPC}-structure A_C is created for a singular belief change C. Its universe consists of all interpretations Ω, all formulas from $\mathcal{L}$, all epistemic states $\mathcal{E}$ as well as the prior and posterior epistemic states and the input formula from C. Formulas are identified by their models because of the assumed syntax-independence. The predicates and functions are interpreted straightforwardly as defined in FO^{TPC}. The full structure

Predicate	Intended meaning	Exemplary appearance
$Mod(w, x)$	w is a model of x	$\omega \in [\![\Psi]\!], \omega \in [\![\alpha]\!]$
$LessEQ(w_1, w_2, e)$	$w_1 \leq w_2$ in e	$\omega_1 \leq_\Psi \omega_2$
$Int(w)$	w is an interpretation	$\omega \in \Omega$
$ES(e)$	e is an epistemic state	$\Psi \in \mathcal{E}$
$Form(a)$	a is a formula	$\alpha \in \mathcal{L}$
Function	**Intended meaning**	**Exemplary appearance**
$op(e_0, a)$	$op(e_0, a)$ is a result of changing e_0 by a	$\Psi \circ \alpha = \Psi'$
$or(a, b)$	propositional disjunction	$Bel\Psi \circ (\alpha \vee \beta) = \ldots$
$not(a)$	propositional negation	$\neg\alpha \notin Bel\Psi \circ \alpha$

Figure 10: Allowed predicates and functions symbols in FO^{TPC}, their intended meaning and how they are typically formulated in belief change literature [SH21]

is described in Figure 11.

Universe	$U^{\mathcal{A}_C} = \Omega \cup \{\Psi_0, \Psi_1\} \cup \mathcal{P}(\Omega)$		
Predicates			
$Mod^{\mathcal{A}_C}$	$= \{(\omega, x) \mid x \in \mathcal{P}(\Omega) \cup \{\Psi_0, \Psi_1\}, \omega \in [\![x]\!]\}$		
$Int^{\mathcal{A}_C}$	$= \Omega$		
$ES^{\mathcal{A}_C}$	$= \{\Psi_0, \Psi_1\}$		
$Form^{\mathcal{A}_C}$	$= \mathcal{P}(\Omega)$		
$LessEQ^{\mathcal{A}_C}$	$= \{(\omega_1, \omega_2, \Psi_i) \mid \omega_1 \leq_{\Psi_i} \omega_2\}$		
Functions			
$or^{\mathcal{A}_C}$	$= \lambda\alpha_1, \alpha_2.\, \alpha_1 \cup \alpha_2$	$e_0^{\mathcal{A}_C}$	$= \Psi_0$
$not^{\mathcal{A}_C}$	$= \lambda\alpha_1.\, \Omega \setminus \alpha_1$	$a^{\mathcal{A}_C}$	$= [\![\alpha]\!]$
$op^{\mathcal{A}_C}$	$= (\{(\Psi, \beta, \Psi) \mid \beta \in \mathcal{P}(\Omega), \Psi \in \{\Psi_0, \Psi_1\}\} \setminus \{(\Psi_0, \alpha, \Psi_0\}) \cup \{(\Psi_0, \alpha, \Psi_1)\}$		

Figure 11: Structure $\mathcal{A}_C$, encoding a singular change $C = (\Psi_0, \alpha, \Psi_1)$ [SH21]

Common ways to define postulates are syntactic postulates (describing changes in belief sets) or semantic postulates (describing changes in a semantic domain, often faithful preorders). FO^{TPC} is a language to describe semantic postulates. Representation theorems exist for most syntactic postulates that connect them to equivalent semantic postulates. Postulates can be expressed as formulas over FO^{TPC}. For example, the success postulate (AGM$*$2):

$$\varphi_{AGM2} = \forall W1((Int(W1) \wedge Mod(W1, op(E0, A)))) \rightarrow Mod(W1, A)$$

A belief change C then satisfies the postulate φ_{AGM2} iff $A_C \vDash \varphi_{AGM2}$ holds. Other

postulates can be represented as formulas over FO^{TPC} in the same way and checked with the same structure A_C.

4.3. A DMF revision operator

Constructing the derived-faithful tpo $\leq_{s_\Psi^M}$ to a DMF system state s_Ψ^M is an important step in the implementation. Algorithm 1 shows a possible algorithm, keeping the order of worlds not in $[\![\mathbb{B}_\Psi]\!]$ aligned to $\preceq_\Psi$ and fixing all worlds in $[\![\mathbb{B}_\Psi]\!]$ in the most plausible rank.

Algorithm 1: Get derived-faithful tpo for DMF system state

Input: DMF System State $s_\Psi^M = (\mathbb{B}_\Psi, \preceq_\Psi)$
Result: Derived-faithful tpo $\leq_{s_\Psi^M}$
$\leq_{s_\Psi^M} = \emptyset$
foreach $(\omega, \omega') \in \preceq_\Psi$ **do**
  if $\omega, \omega' \notin [\![\mathbb{B}_\Psi]\!]$ **then**
    $\leq_{s_\Psi^M} := \leq_{s_\Psi^M} \cup \{(\omega, \omega')\}$
  end
end
foreach $\omega \in [\![\mathbb{B}_\Psi]\!]$ **do**
  $\leq_{s_\Psi^M} := \leq_{s_\Psi^M} \cup \{(\omega, \omega') \mid \omega' \in \Omega - [\![\mathbb{B}_\Psi]\!]\}$
end

With the ability to compute derived-faithful tpos to DMF system states, a DMF revision operator according to Definition 11 can be implemented as follows:

1. Construct $\leq_{s_\Psi^M}$ for the initial DMF system state s_Ψ^M, encoding a revision operator $*_{s_\Psi^M}$ as described in Definition 10
2. Compute $\mathbb{B}_{\Psi\circ\alpha} = \mathbb{B}_\Psi *_{s_\Psi^M} \alpha$
   a) The minimal models of the input α in $\leq_{s_\Psi^M}$ define the belief set of $\Psi \circ \alpha$: $[\![Bel(\Psi \circ \alpha)]\!] = min([\![\alpha]\!], \leq_{s_\Psi^M})$
   b) Set $\mathbb{B}_{\Psi\circ\alpha} \coloneqq th(min([\![\alpha]\!], \leq_{s_\Psi^M}))$
3. Find $\preceq_{\Psi\circ\alpha}$
   a) Find an outgoing edge e from the context tpo $\preceq_\Psi$ in M over a formula that is semantically equivalent to α: $(\preceq_\Psi, \beta, \preceq')$ with $\beta \equiv \alpha$
   b) If e exists, set $\preceq_{\Psi\circ\alpha} := \preceq'$, otherwise $\preceq_{\Psi\circ\alpha} := \preceq_\Psi$
4. Set $s_{\Psi\circ\alpha}^M \coloneqq (\mathbb{B}_{\Psi\circ\alpha}, \preceq_{\Psi\circ\alpha})$

4.4. User Experience

Any software that is meant to be used for empirical research should provide an intuitive user experience. This is particularly true for tools that can be used in a didactic setting to explain complex concepts to students.

During the automated certification of postulates in belief change, a user has to configure multiple inputs. Firstly, some form of initial agent state and new information that the agent learns. If no automated belief revision is implemented, a user also has to provide the posterior agent state by hand. The tool Alchourron, described in [SH21], takes this approach. Users can configure a prior and posterior agent state on a single page. This allows the tool to abstract from a concrete belief change operator but places more configuration work on the user. The approach works for Alchourron because the chosen instantiation of an epistemic state is a simple tpo over Ω, and it only handles a single belief revision.

In contrast to Alchourron, the goals for this work were to automate DMF revision and allow for multiple belief change inputs. Additionally, the instantiation of epistemic states with DMF system states is not as simple as with a tpo. Users will need to configure a DMF system M and an initial DMF system state. Result display will also be more complex because the certification of postulates happens for every configured belief change step.

Because of the added complexity, it will make sense to break up the user interface into multiple sections. At the very least, the configuration and result display sections should be split. Relevant information from previous steps should still be available on-demand (e.g., the underlying DMF system M will stay relevant while configuring a DMF system state) but hidden as long as the user does not need it. For the results, a filter for postulates that are satisfied by all configured belief changes would allow for a quick overview.

Most importantly, it will be helpful to discuss the interface with actual users of the tool and iterate on it with feedback.

5. Coeus: Implementation Details

An implementation to support future research on DMF revision is the primary goal of this work. The resulting software is named Coeus after the greek titan representing rational intelligence, Κοιος (meaning query or questioning). Coeus is available online at `http://coeus.rhazn.com`. Most components are open source and can be retrieved digitally. In addition, the digital media accompanying this work contains all source code (as described in Appendix A).

Here, Coeus will refer to the complete software as found at `http://coeus.rhazn.com`, consisting of a backend server written in Java and a browser-based frontend built on TypeScript.

The implementation will be presented by first outlining identified challenges. A walk-through of the developed tool is included as context for the following chapters. They present an overview of the architecture and details on every module, from the server-side API to the browser-based frontend. Solutions to major challenges like textual representation are presented in their own chapters when appropriate. The algorithm to implement a DMF revision operator itself closes the chapter.

5.1. Implementation challenges

The stated goals from Subsection 4.1 pose a variety of challenges for the realization of an application. They can be divided into technical challenges for the implementation and approaches to communicate the dense amount of information to users.

On the technical side, in addition to an implementation of generic first-order logic, support for common tools from belief revision like total preorders is needed. For this reason, the tool presented here extends an existing institutional library called *logical-systems*, written in Java. A possible solution to making the software easy to share and install would be to provide it as a browser-based application. At the time of writing, no adequate library to work with belief revision was available for browsers. Providing own implementations and interacting with the existing implementation in logical-systems will therefore be a challenge.

A general problem for browser-based applications is the quick rate of change for standards and frameworks. In addition, with browser-based applications becoming more and more complex, a number of frameworks were created, leading to incompatibilities of implementations between them. Careful attention to modularization and good software practices will be needed to ensure the tool can be supported with future changes.

Furthermore, automated certification of postulates for belief changes is a complex problem. The approach using model-checking is prone to performance problems with larger input sizes. The existing implementation is hard to extend because postulates are implemented directly in Java and can not be changed without recompilation and deployment.

Therefore, the topic of textual representation is relevant to extend logical-systems with new postulates. A format for a formula must be chosen to load new postulates from files or allow users to provide their own. No clear standard is available but various options for a syntax like TPTP [Sut17], InfoCF [Kut19] or TeX exist. Likewise, total preorders are part of DMF revision and have to be encoded as text to support exporting and importing Coeus configurations. Similar to formulas, no clear standard exists for a textual representation of total preorders. Because of their growth in size with larger signatures, any solution will also need to be efficient.

As discussed in the previous chapter, communicating complex information to users will also be a challenge. Coeus will need to be carefully broken up into smaller parts not to overwhelm researchers or students. A special focus has to be applied to providing auxiliary information in a way that does not interfere with the main flow of configuring belief revision steps.

5.2. Coeus Walkthrough

A short introduction of the final implementation is presented here. All screenshots are showing the tool configured for the DMF belief revision described in Example 3. The configuration was exported and can be downloaded from the website.

Coeus allows users to complete the following steps:

1. Define a signature Σ or import a previous configuration
2. Create a DMF system M
3. Configure the initial DMF system state s^M_Ψ
4. Set up one or multiple inputs for (iterated) belief revision
5. See a list of postulates from the literature and if they are fulfilled or not for each belief revision step, optionally test their own postulate
6. Optionally export their work

To begin, the user has to configure a signature Σ or can optionally import previous work. Figure 12 shows the currently defined signature and the ability to configure a new signature using the slider.

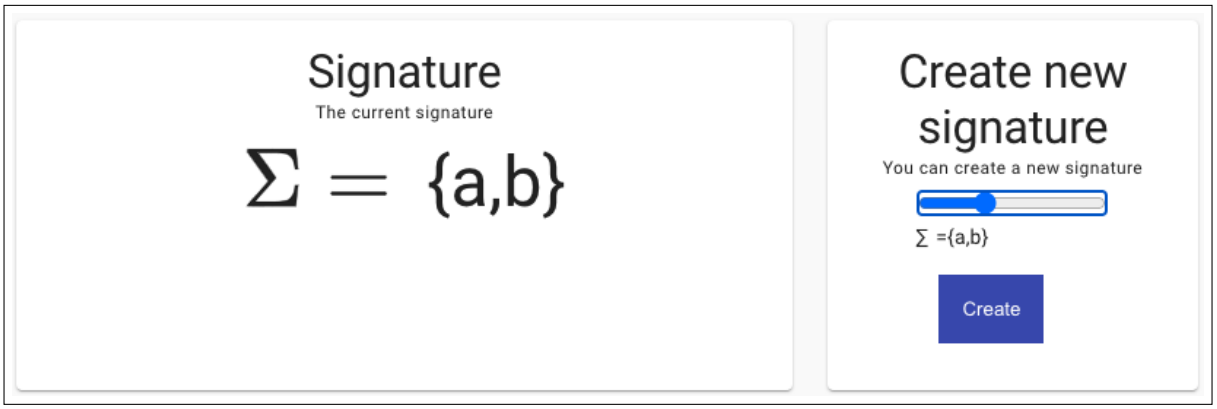


Figure 12: Configuring a signature

The next screen allows the user to set up a DMF system M in two ways, either as a list of nodes and edges (Figure 13) or as a graph (Figure 14). In any case, the individual tpos and edges can be selected and changed on the right-hand side.

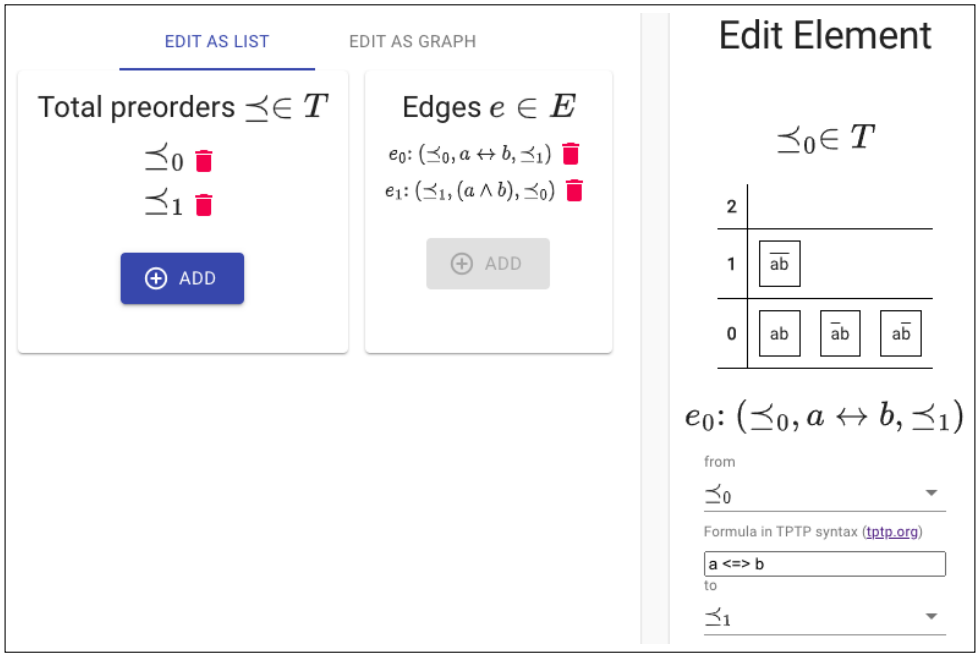


Figure 13: DMF system configuration as list

The initial DMF system state s_Ψ^M of an agent is defined next. Figure 15 shows the screen that displays the current state on top and shows a dropdown to select the initial

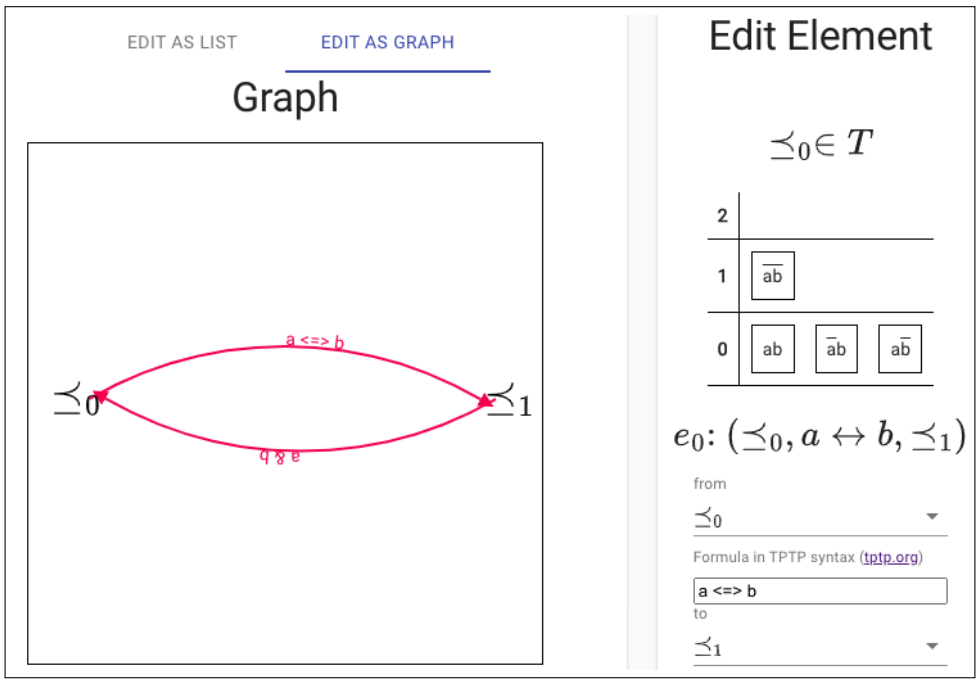


Figure 14: DMF system configuration as graph

context tpo from M as well as input fields for formulas in the belief set.

After configuring the initial state of the system (by defining the DM system M and the initial DMF system state of the agent s_Ψ^M), the user can create one or multiple belief revision inputs. Figure 16 shows the belief revision input. The current DMF system state s_Ψ^M is shown in the left box (along with overlays for its derived-faithful tpo $\leq_{s_\Psi^M}$ and the context tpo from M). When the user enters an input formula α in the center, Coeus automatically computes the posterior DMF system state $s_{\Psi \circ \alpha}^M$ and shows it on the right side.

Belief revision steps can be saved by pressing "Perform Revision" and are then displayed, in reverse order, below the input as shown in Figure 17.

The final screen of Coeus is the results panel. The settings (see Figure 18) allow the user to define their own postulate to certify, filter postulates by name, or only show fulfilled postulates.

Results are shown in two ways. A result overview (shown in Figure 19) displays all belief revision steps in one table and allows the user to see which revision inputs are satisfying what postulate. For a more detailed view, each individual input is displayed as shown in Figure 20. In this view, the user can also check the derived-faithful tpos and context tpos again and see the formulas for the certified postulates.

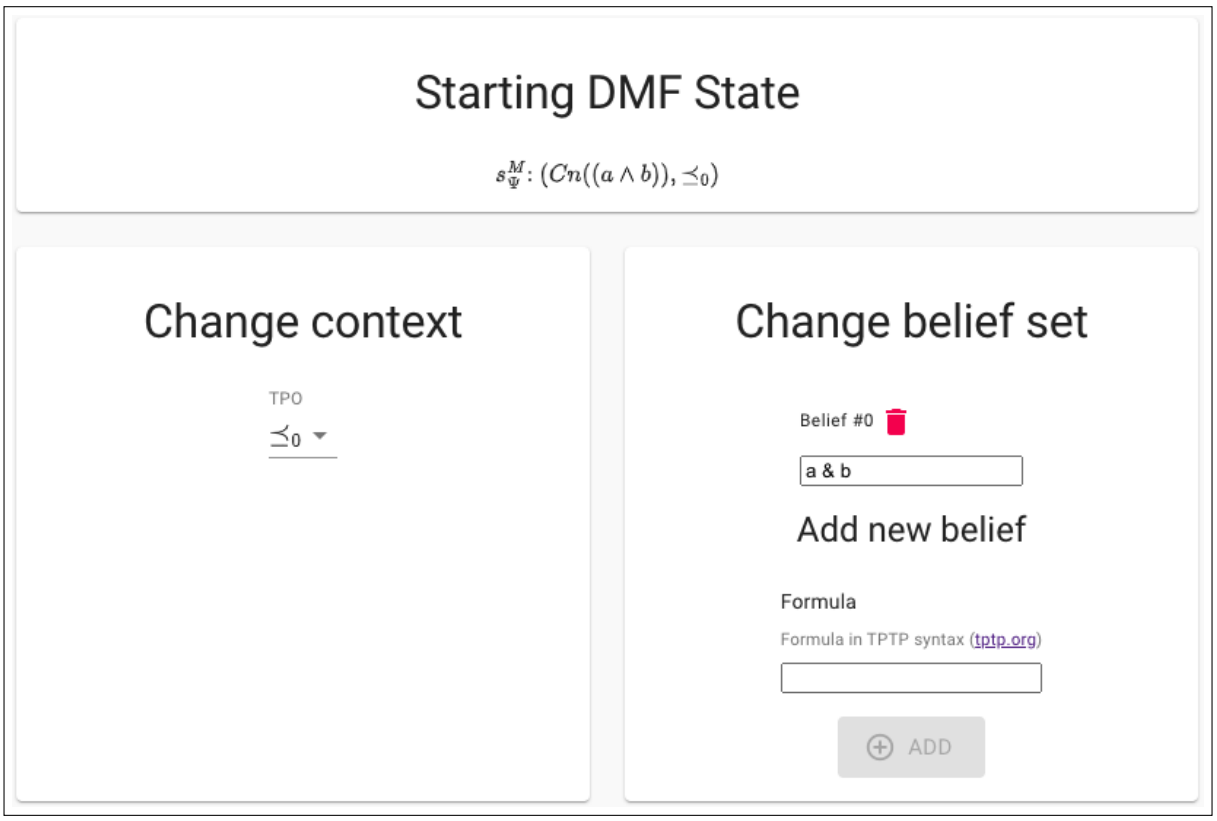


Figure 15: Initial DMF system state

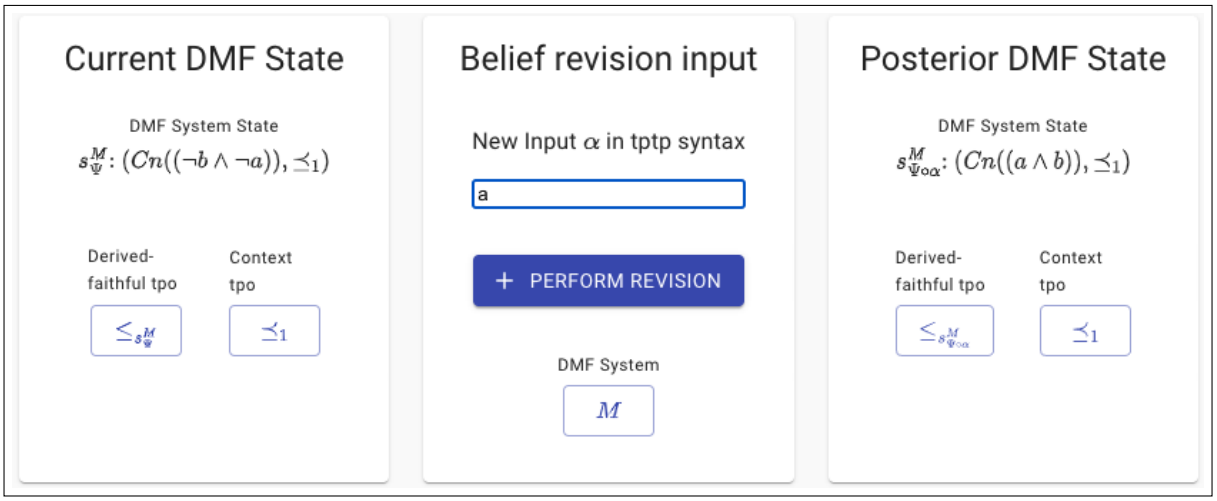


Figure 16: Input for a belief revision step

5.3. Architecture Overview

To achieve the goals defined in Subsection 4.1, Coeus is implemented in a Client/Server architecture using a browser-based frontend. The choice allows for easy distribution and

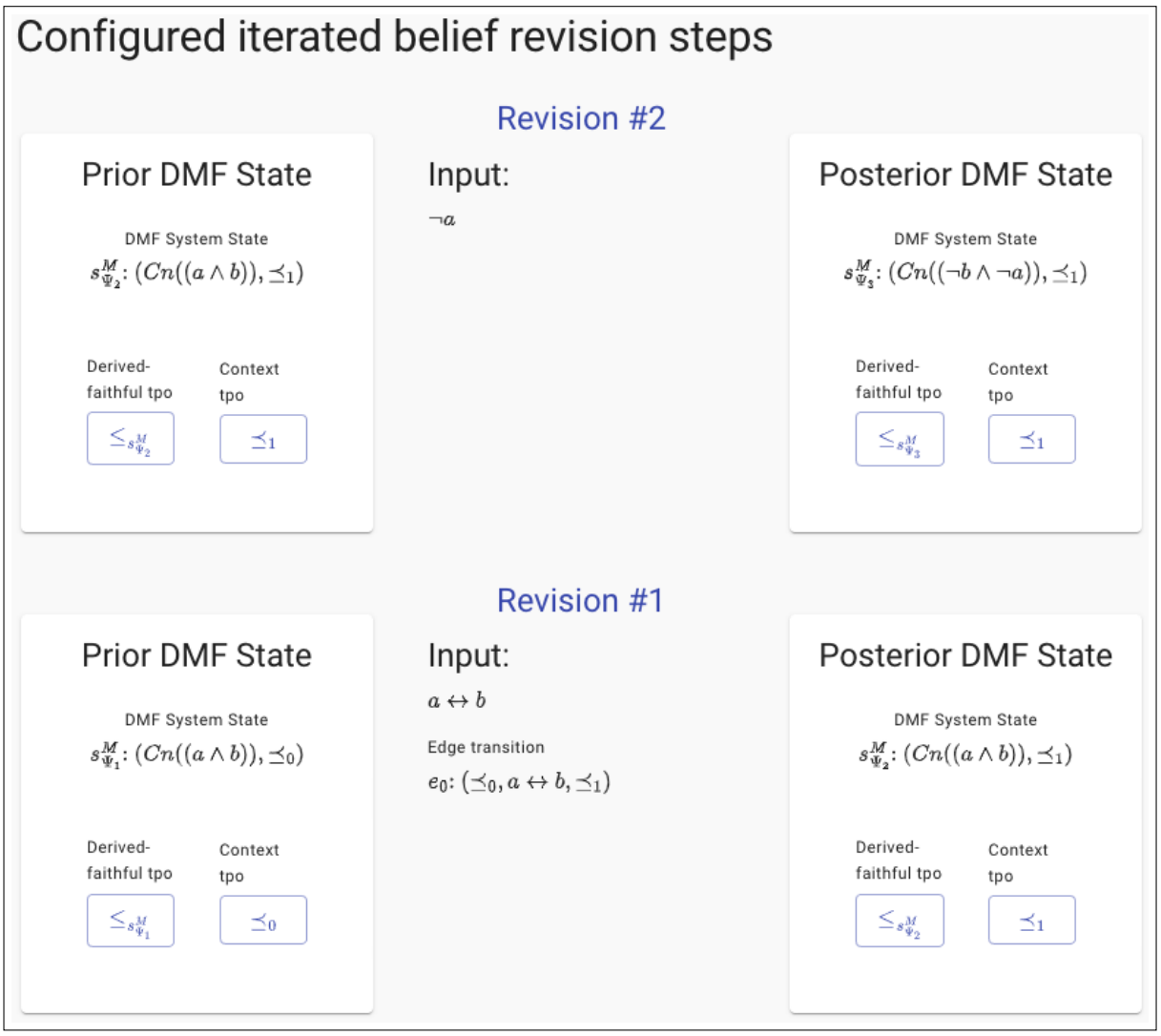


Figure 17: List of iterated belief revisions

setup of the tool while still using the increased computational power of a server.

The majority of logic and calculations are done using an existing logic library called *logical-systems* [Sau21], written in Java. The server-side library provides the ability to verify whether a belief change, expressed with a prior and posterior tpo as well as an input formula, satisfies a given postulate or not. In addition, it was extended with functionality to load postulates from a textual representation.

The library was exposed to clients with an additional Java project (called postulate-check-server) that provides a set of HTTP endpoints. It allows clients to use some functions of the underlying library over HTTP as well as implements various custom endpoints to support DMF revision.

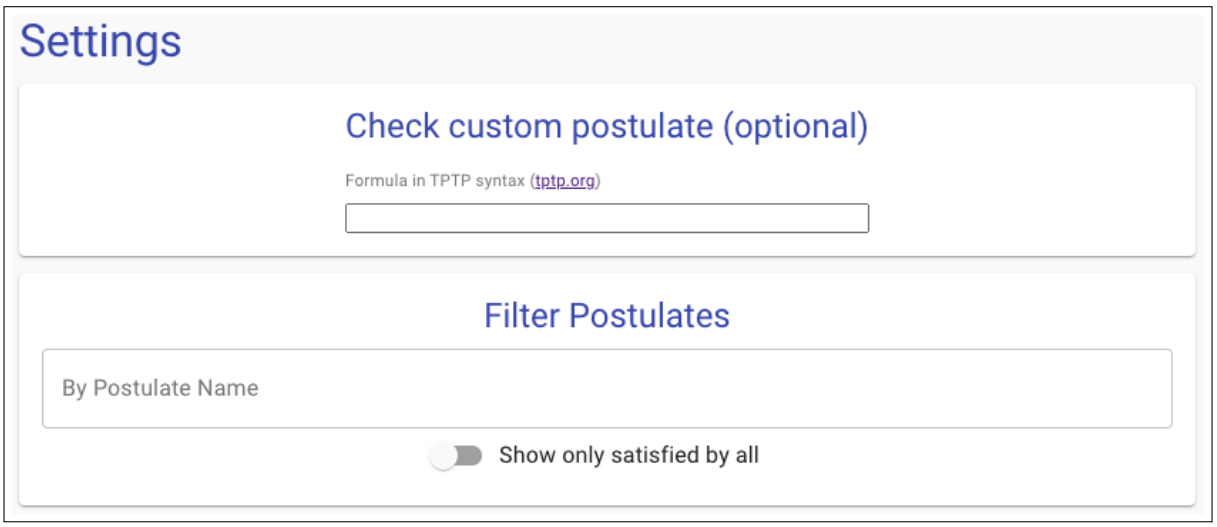


Figure 18: Settings for result screen

Figure 19: Results of all revision steps as overview

Finally, a web-based client was written in TypeScript. TypeScript was chosen for its strict type system in contrast to relying only on Javascript. The frontend consumes the API endpoints provided by the server and adds an interface for users to configure DMF systems, DMF system states, and belief revision inputs. In addition, it allows users to export and import their work.

The high-level architecture is shown in Figure 21.

A modular approach was chosen for the frontend to increase reusability. The final

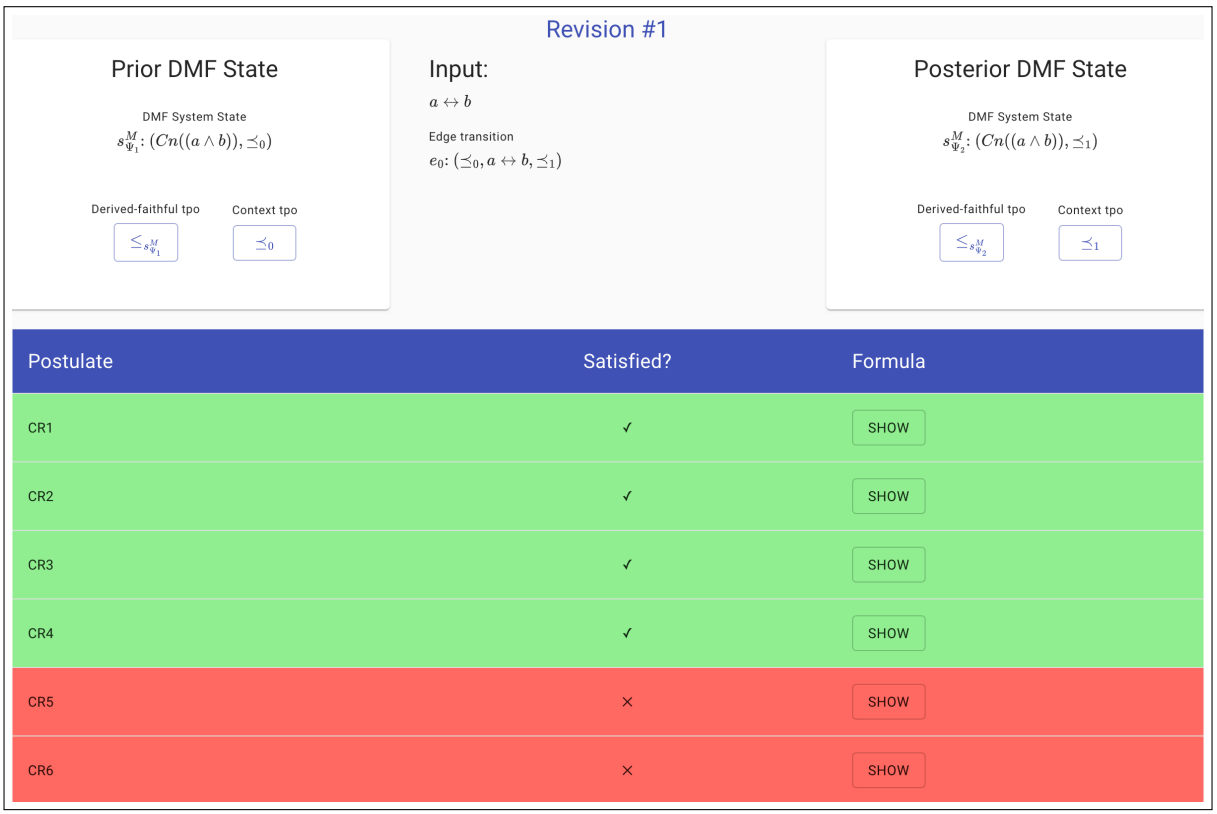


Figure 20: Detailed results for each revision step

Coeus frontend is built from the following packages:

1. *logic-ts* [Hel21b] implements basic building blocks of propositional logic in TypeScript. It contains classes to represent e.g. propositional worlds or signatures and their (de-)serialization.

2. *logic-components* [Hel21a] builds on *logic-ts* by adding a visualization layer. It defines components to display and interact with, e.g., signatures or tpos in any modern browser with the goal of providing building blocks for future tools.

3. *dmf-revision* is a frontend implemented using components from *logic-components* to provide a custom interface for DMF revision.

Figure 22 shows an overview of the package structure. Because of the modular approach *logic-components* could also be used to implement other tools than *dmf-revision*, namely Alchourron ([Sau21, SHB21]) as well as Preference Builder (a website to customize preferences over worlds and export them as text [Hel21c]).

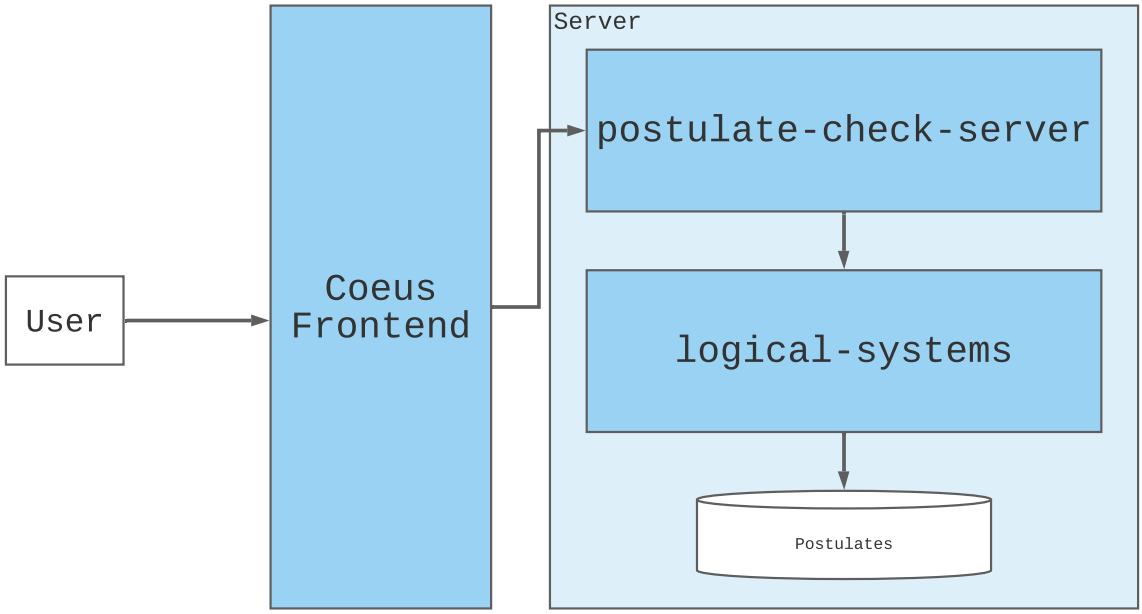


Figure 21: Software Architecture

5.4. Textual representation

A problem during the development of Coeus was how to represent various elements of the system as strings. Being able to export and import DMF systems and DMF system states supports the goal of easy sharing. In addition, the HTTP protocol used for Client/Server communication is text-based, so any entities that are exchanged between the front- and the backend need to be serialized as well.

Because of that, formulas and tpos require a textual representation. Export and import of DMF systems and DMF system states can reuse the existing logic for formulas and tpos. For reference, a full export of Example 3 can be downloaded from the website at `http://coeus.rhazn.com`.

Formulas

A subset of the TPTP ("Thousands of Problems for Theorem Provers") syntax [Sut17] was chosen to represent propositional formulas internally. The TPTP project provides a library of test problems for automated theorem proving (ATP). Because the TPTP syntax is commonly used to describe problems for ATP, a selection of parsers is available. The TPTP syntax was also chosen for formula input to have a high chance that users would be familiar with the language.

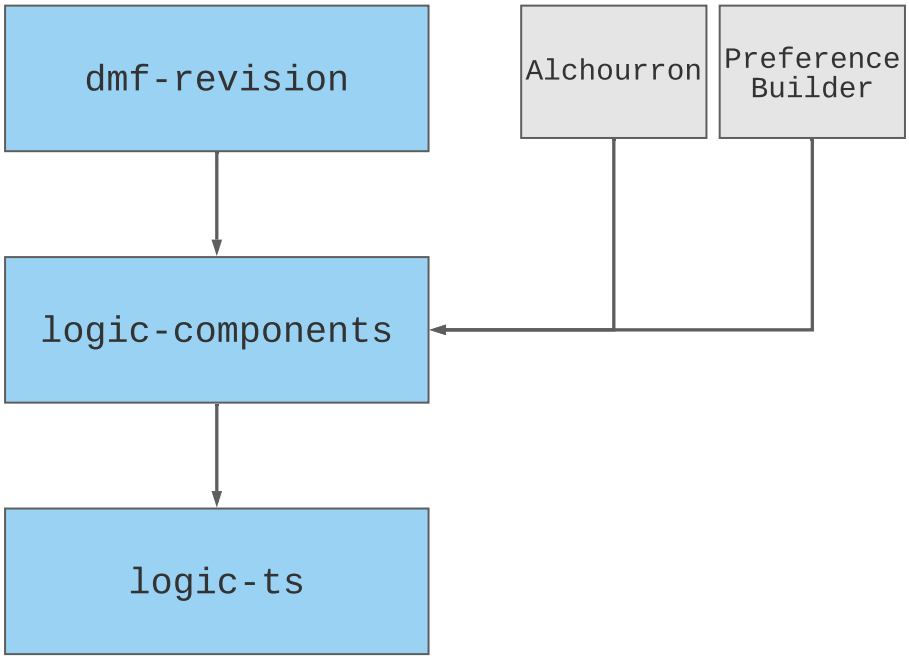


Figure 22: Package structure of Coeus (in blue)

Outside of internal usage and user input, Coeus represents formulas using TeX syntax. In the browser, the formula is rendered as an SVG image using MathJax. The backend provides endpoints to exchange a formula in TPTP syntax with its TeX equivalent. Using different textual representations was chosen as a compromise as it allows the implementation to reuse libraries and work with existing standards. On the other hand, when formulas are displayed to be read by humans, they are formatted in the familiar style of mathematical notation.

World Preferences

In order to support a wider range of use cases, the decision was made to support not only the serialization of tpos but also ordinal conditional functions [Spo88]. Ordinal conditional functions are a common knowledge-representation formalism. Both ocfs, as well as tpos, can be represented as a ranking of worlds. In this context, in contrast to tpos, ocfs allow empty ranks. By allowing empty ranks, worlds are not only ordered relative to each other but also assigned a quantitative value of plausibility. Figure 23 shows such a ocf on all worlds of $\Sigma = \{a, b\}$. logic-components provides a component to configure tpos as well as ordinal conditional functions based on the concept of assigning ranks to worlds.

$$\begin{array}{c|l} 2 & ab,\ \overline{a}\overline{b} \\ 1 & \\ 0 & a\overline{b}, \overline{a}b \end{array}$$

Figure 23: Ordinal conditional function on $\{ab, a\overline{b}, \overline{a}b, \overline{a}\overline{b}\}$

As tpos can be thought of as ocfs without empty ranks, any formalism that can represent an ocf can also represent a tpo. Therefore, three approaches for a textual representation of ocfs are supported and compared here.

Because logic-ts is implemented in TypeScript an intuitive representation is JSON. Simple TypeScript types can be easily serialized and parsed using the build in `JSON.stringify` and `JSON.parse` methods. In addition, JSON is a human-readable format that allows users to edit the generated textual representation directly.

To represent a propositional world ω as JSON, logic-ts serializes an array of all propositional variables that are *true* in ω, e.g. $\omega = a\overline{b}c$ is represented as `["a", "c"]`. An ocf can be encoded as an ordered list of ranks with the ranks themselves encoded as list of worlds. Listing 1 shows a textual representation of Figure 23 as JSON.

```
{
    "signature": ["a", "b"],
    "ranks": [
        [
            ["a"],
            ["b"]
        ],
        [],
        [
            ["a", "b"],
            []
        ]
    ]
}
```

Listing 1: Ocf represented as JSON

While human-readable and easy to implement, the JSON approach is limited by size. For total preorders, the number of worlds that have to be encoded grows exponentially with signature size (as $2^{|\Sigma|}$). In addition, large numbers of empty ranks (as can exist in

ocfs) are still encoded normally.

An alternative implementation in logic-ts with lower space requirements is directly working with binary data. Two different options are explored here. The *worldlist* approach is similar to the JSON representation and saves every rank as a list of worlds. The opposite view is chosen in the *ranklist* approach that assigns a number to every world and saves an ordered list of numbers for their rank.

For the worldlist approach, any world can be encoded in binary with $|\Sigma|$ bits by ordering the propositional variables alphabetically and assigning 1 to *true* variables and 0 to others, e.g. $\omega = a\overline{b} = 10$. Every world in Ω can be assigned a natural number, called world index, using this numbering schema.

TypeScript allows for managing binary data in 8, 16 and 32 bit increments so the lowest possible size to encode all worlds is automatically chosen (e.g. 8 bits for any $|\Sigma| \leq 8$). To reduce the amount of data required for empty ranks, only ranks that contain worlds are considered. A rank is encoded as two 32-bit numbers. The first number contains the number of worlds in the rank n, the second the distance to the next non-empty rank d. During parsing, a rank can be restored by interpreting the next $|\Sigma| * n$ bits as worlds. After restoring the rank, a number of empty ranks equal to the distance d are created. Listing 2 shows the ocf from Figure 23 encoded as binary in the worldlist approach. The leading 8-bit number encodes a version (in this case version 1). Because the size of the alphabet Σ is relevant to compute the size of worlds the second, 32-bit, number encodes $|\Sigma|$ (in this case three). The full structure until the first rank is:
`version:alphabet-size:#worlds-in-rank:distance:world1:world2:....`

```
00000001                                // Version: 1
00000000000000000000000000000010 // Signature size: 2

00000000000000000000000000000010 // Rank size: 2
00000000000000000000000000000001 // Gap size: 1
00000010                                // World 2
00000001                                // World 1

00000000000000000000000000000010 // Rank size: 2
00000000000000000000000000000000 // Gap size: 0
00000011                                // World 3
00000000                                // World 0
```

Listing 2: Ocf in worldlist binary

When encoding ocfs in the ranklist approach, the binary representation is an ordered list of rank numbers assigned to the possible world at that index. The world index corresponds to its binary number as described for the worldlist approach. For example the index 2 holds the rank for world $\omega = a\bar{b} = 10$.

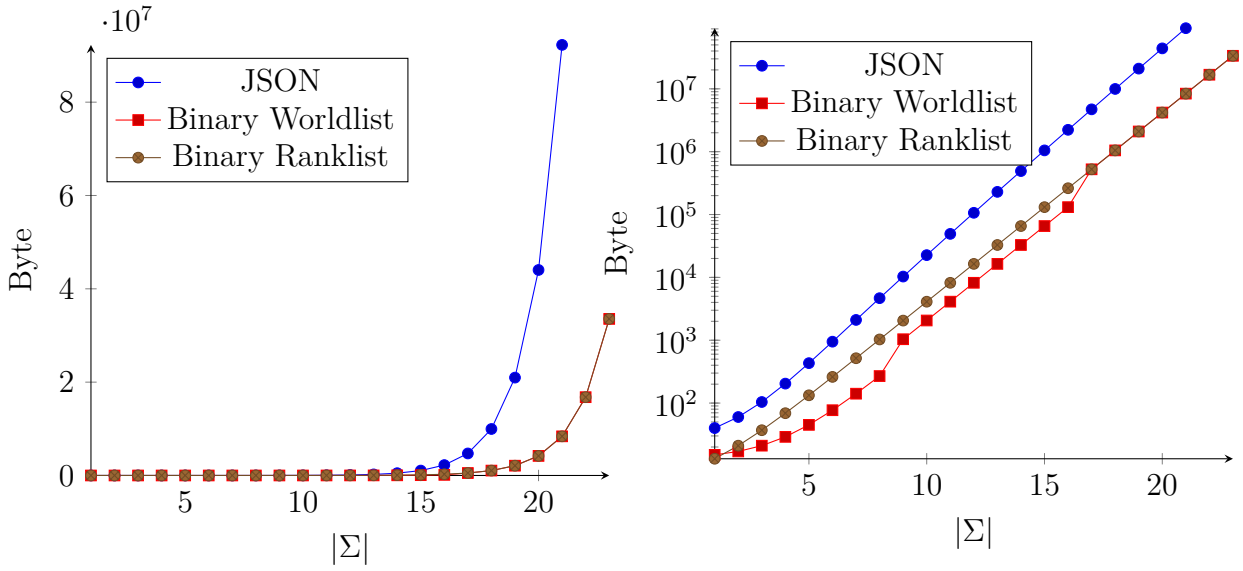
Listing 3 shows the ocf from Figure 23 encoded as binary in the ranklist format with the same leading data encoding the version and the alphabet size.

```
00000001                                // Version: 1
00000000000000000000000000000010 // Signature size: 2

00000000000000000000000000000010 // World 0 is on rank 2
00000000000000000000000000000000 // World 1 is on rank 0
00000000000000000000000000000000 // World 2 is on rank 0
00000000000000000000000000000010 // World 3 is on rank 2
```

Listing 3: Ocf in ranklist binary

To compare how the approaches scale, Figure 24 shows the size of a tpo with a single rank depending on the size of Σ in bytes. Note that this comparison does not create empty ranks and is therefore skewed in favor of the JSON and ranklist approaches. The jumps in size of the worldlist binary encoding at $|\Sigma| = 9$ and $|\Sigma| = 17$ are due to the



(a) Size of serialized tpo (linear)

(b) Size of serialized tpo (log)

Figure 24: Size comparison of a serialized with a single rank

switch to a bigger data type to encode worlds. The ranklist binary representation uses an unsigned 32-bit integer (for a maximum rank of 4294967295). While a smaller maximum rank would lead to a smaller absolute size, the relative scaling of the approach would stay the same as it is related to the number of worlds.

In local testing, the JSON approach was able to encode the tpo up to $|\Sigma| = 21$ and used 92274800 bytes (in worldlist binary, the same size encodes in 8388621 bytes or at 9% of the size). The wordlist binary solution was able to encode $|\Sigma| = 23$ at 33554445 bytes (33554437 bytes for the ranklist approach). All approaches are likely limited by memory constrains on the test machine, but the difference in scaling is visible.

Fundamentally, all approaches are limited by scaling based on the number of possible worlds that have to be represented. When considering total orders, $2^{|\Sigma|}$ possible worlds must be either be encoded directly or a rank number assigned to each one. Both the JSON approach and the worldlist binary approach have reduced sizes for partial orders. Because of the rapid increase in the number of worlds, binary solutions scale better as their representation of an individual possible world is more compact. On the other hand, the JSON representation is human-readable and easier to implement, making it a good choice for smaller signatures. Coeus exports tpos as JSON because they are created from user input, ensuring the number of possible worlds is relatively small.

5.5. Extending logical-systems

logical-systems [Sau21] is a modular logic-library with support for a variety of different logics. It implements a solution to the problem of verifying whether or not a belief change satisfies a set of postulates or not by employing first-order model checking as described in Subsection 4.2.

In order to make it easy to extend the system, postulates are loaded from a textual representation of formulas in TPTP syntax. For the prototype, several postulates from the literature were implemented, described in full in Appendix C. As an example, Listing 4 shows a textual representation of φ_{AGM2}. Because postulates can be provided as simple FO^{TPC} formulas, the system will be easy to extend in the future. Additionally, this approach allows users to test custom postulates by entering them in TPTP syntax.

```
fof(
    'Success',
    postulate,
    ! [W1] : ( ( int(W1) & mod(W1,op(E0,A)) ) => mod(W1,A) )
).
```

Listing 4: (AGM∗2) represented as a formula in FO^{TPC} using TPTP syntax

A custom connector to a scala-based parser (scala-tptp-parser [Ste21]) was developed to construct the internal representation of a formula needed in the backend. Together with the author, the parser itself was enhanced to be available from widespread package managers for both Scala and Java [1]. A set of unit tests that parse existing formulas from the TPTP library was also added. For any parsed node from the scala-tptp-parser, the connector creates the corresponding object in *logical-systems*. Different configurations for these connections can be defined and exchanged during runtime. The postulate-check-server uses this flexibility to provide two endpoints to check a formula, one for first-order logic and one for propositional logic.

In summary, *logical-systems* provides the core functionality to work with first-order and propositional logic formulas as well as to verify whether a belief change fulfills a postulate or not.

[1] `https://github.com/leoprover/scala-tptp-parser/pull/1`

5.6. Postulate check server

postulate-check-server is a Java application that provides a HTTP API for Coeus. It is based on *logical-systems* and enables any frontend to use features such as checking logical formulas for equivalence, minimizing propositional formulas using the Quine–McCluskey algorithm [McC56] or certifying if a belief change fulfills postulates using model checking [Sau21, SHB21].

Postulate-check-server is built with Spark [2], a microframework for creating web applications. Application logic is grouped into services that are provided to controllers by dependency injection [Fow04]. On application start, HTTP routes are matched to controller functions. In addition to the implementations relevant for belief change, the app also handles general functionality like HTTPS. For transforming input and output from Java objects to JSON, GSON [3] is used. Figure 25 shows a high-level overview of the architecture.

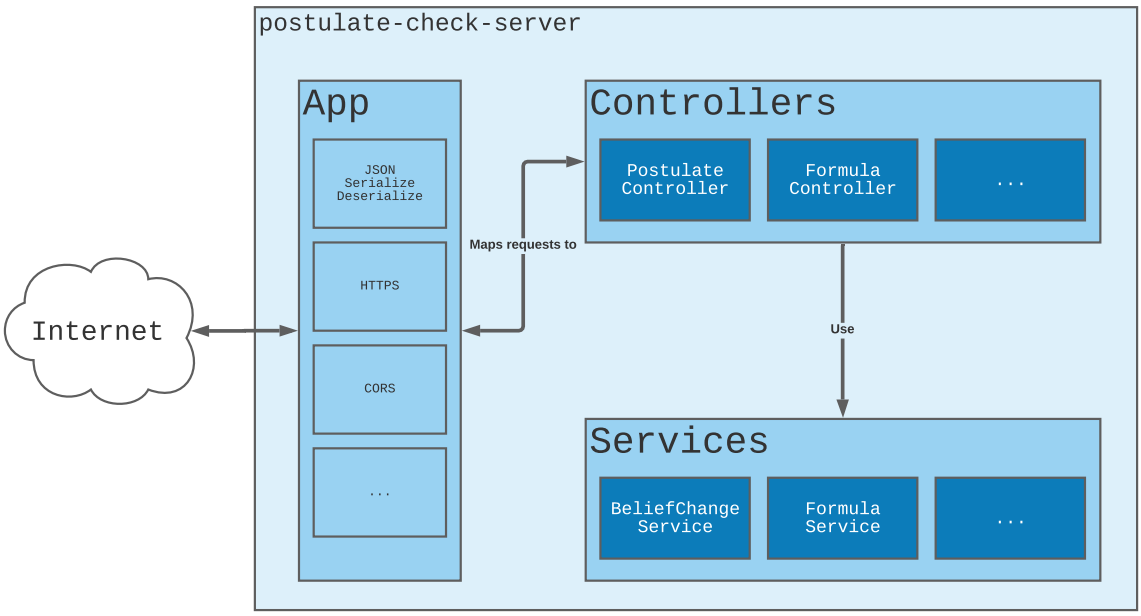


Figure 25: postulate-check-server architecture

Listing 5 is an example for a controller implementation that returns the LateX representation of a formula in TPTP syntax. A GSON instance is provided using constructor injection in line 4 and used to deserialize the HTTP input in line 9. Using the Spark

[2] https://sparkjava.com/
[3] https://github.com/google/gson

framework, the function `formulaToLatex` is registered as a callback for HTTP requests and provides an API for functionality in the logic-components library.

```
public class FormulaController {
  private final Gson gson;

  public FormulaController(Gson gson) {
    this.gson = gson;
  }

  public Object formulaToLatex(Request request, Response response) {
    FormulaToLatexInput input = gson.fromJson(request.body(),
        FormulaToLatexInput.class);

    Formula<PropositionalSignature<Character>> formula =
        FormulaParseService.propositionalFormulaFromString(
         input.formula);

    FormulaToLatexOutput result = new FormulaToLatexOutput();
    result.latex = formula.toLaTeX();

    return result;
  }
}
```

Listing 5: Excerpts from the FormulaController of postulate-check-server (`src/main/java/com/rhazn/controller/PostulateController.java`)

To enable GSON to automatically serialize HTTP responses to JSON, custom serializers were implemented for objects in logcial-systems. The class `InterpretationSerializer` (see Listing 6) is an example that translates a propositional interpretation into the JSON structure described in Subsection 5.4.

```java
public class InterpretationSerializer implements JsonSerializer<
    PropositionalInterpretation<Character>> {
  @Override
  public JsonElement serialize(
      PropositionalInterpretation<Character>
          propositionalInterpretation,
      Type type,
      JsonSerializationContext jsonSerializationContext
  ) {
    JsonArray truthyValues = new JsonArray();

    for (Character variable : propositionalInterpretation.getAllTrue())
        {
      truthyValues.add(variable);
    }

    return truthyValues;
  }
}
```

Listing 6: Excerpts from the InterpretationSerializer of postulate-check-server (`src/main/java/com/rhazn/json/InterpretationSerializer.java`)

5.7. Coeus client

A browser-based client was implemented to interact with the API provided by postulate-web-server. It is built highly modularized and allowed for component reuse across multiple projects during the writing of this work. This chapter will give an overview of the goals for the Coeus client, approaches to manage software complexity due to modularization, and then discuss the individual modules in more detail.

FAIR principles

As described in Subsection 4.1, a goal during the implementation of Coeus was to follow the FAIR principles for good research software. The complete implementation is

available on GitHub[4], the biggest platform for collaborative software development in the world [BHV16]. In addition, every module written solely by the author is provided with a permissive open source license. While it is easy to find and use code from GitHub for developers, it does not provide solutions for some goals for research software, namely making it easy to cite and download exact software snapshots. For this purpose, the modules are also uploaded to the repository Zenodo [EO13], providing a citeable digital object identifier (DOI) as well as the ability to download software snapshots.

The goal of targeting as many runtime platforms as possible was achieved by making Coeus a web-based application that can be accessed with any modern browser. This ensures the software is easy to distribute and update as well as making it widely available.

Special attention was paid to the usage of standards to ensure interoperability and reusability. Export of configuration is done using JSON (Subsection 5.4), a textual representation that is human-readable and is commonly used. Modularity and independence from any concrete frontend framework for most of the logic implementation was a particular concern. By splitting up the implementation into multiple packages, large parts of it could be implemented in TypeScript only or employ HTML standards like custom elements.

NPM Modules

As described in Subsection 5.3 and shown in Figure 22, the Coeus client is split into three packages, logic-ts (logic entities themselves) [Hel21b], logic-components [Hel21a], (web components to manipulate and display logic entities) and dmf-revision (an implementation of postulate checking for DMF revision). A fourth package, *logic-components-react*, exists to allow for easier use of the components from logic-components in React based applications.

One option to manage dependencies in Javascript applications is npm [5]. npm is a package manager that can resolve dependencies from a registry, typically the public npm registry.

Splitting up software over multiple modules introduces a considerable overhead in maintenance. Therefore, it was crucial to automate as much as possible. For this purpose Github Actions [6] were used to implement continuous integration and -deployment. Originally a practice from extreme programming [Bec99], continuous integration recom-

[4] `https://github.com`
[5] `https://www.npmjs.com/`
[6] `https://github.com/features/actions`

mends automating software builds and ensuring changes do not introduce mistakes by including tests. Continuous deployment is the practice of deploying software artifacts from successful builds automatically. In the case of the npm modules discussed here, that means uploading a new version of the module automatically whenever their version number changes. Listing 7 shows a configuration file for Github Actions that sets up continuous integration and -deployment for logic-ts. Line 18 executes tests defined in the project after which the project is built. After a successful test and build, the artifact is published to npm using the secret configured in the repository (line 22). Similar automated build scripts exist in all package repositories.

```
1  name: CI
2
3  on:
4    push:
5    branches:
6      - master
7
8  jobs:
9    publish:
10   runs-on: ubuntu-latest
11   steps:
12     - uses: actions/checkout@v2
13     - uses: actions/setup-node@v2
14     with:
15       node-version: 14.16.0
16     - run: npm install -g npm@7.8.0
17     - run: npm install
18     - run: npm run test:ci
19     - run: npm run build
20     - uses: JS-DevTools/npm-publish@v1
21     with:
22       token: ${{ secrets.NPM_ACCESS_TOKEN }}
23       access: public
```

Listing 7: Continuous Integration in logic-ts using Github Actions (`/.github/workflows/ci.yml`)

logic-ts

As the foundation for implementations of propositional logic in TypeScript, the logic-ts package implements various logical entities.

The textual representation of logical entities that were discussed theoretically in Subsection 5.4 was implemented in logic-ts. Methods for the serialization into a textual representation are provided by the classes themselves. Depending on which representation they offer, classes implement either the interface `SerializeableBinary.ts` or `SerializeableBinary.ts` in `src/serialize/interface/`. Both interfaces are Client/Server-Interfaces, as defined by Steimann und Mayer [SM05]. The implementation directly on the class allows to call, e.g., a `.toJson()` method on any object that implements the correct interface to retrieve a textual representation of the object.

To parse a text and recreate the object it represents, parser classes that implement either a `ParserFactoryBinary` or `ParserFactoryJson` interface exist. logic-ts implements parsers for most entities at `src/serialize/`. Listing 8 shows an example of such a parser for a propositional world. It supports parsing a world from either a list of truthy values in JSON representation or a binary number as described in Subsection 5.4.

```typescript
export class PropositionalWorldParserFactory
  implements ParserFactoryJson<PropositionalWorld>, ParserFactoryBinary
     <PropositionalWorld>
{
  constructor(private signature: PropositionalSignature) {}

  public fromJson(json: string): PropositionalWorld {
    const parsed = JSON.parse(json);

    return new PropositionalWorld(this.signature, new Set(parsed));
  }

  public fromBinary(binary: ArrayBuffer): PropositionalWorld {
    const view = PropositionalWorldParserFactory.
       getMinimalViewForSignatureSize(this.signature.size, binary);

    return PropositionalWorldParserFactory.worldFromNumber(this.
       signature, view[0]);
  }

  public static worldFromNumber(signature: PropositionalSignature,
     number: number): PropositionalWorld {
    const assignment = [...number.toString(2).padStart(signature.size,
       "0")].reduce((previous, current, index) => {
      return previous.concat(current === "1" ? [...signature][index] :
         []);
    }, []);

    return new PropositionalWorld(signature, new Set(assignment));
  }
}
```

Listing 8: Excerpts from PropositionalWorldParserFactory (`src/serialize/PropositionalWorldParserFactory.ts`)

Serialization and parsing in logic-ts are extensively unit tested using jest [7]. Unit tests can be found in the `src/test` directory and are automatically executed on every commit

[7] `https://jestjs.io/`

to verify its correctness. Listing 9 shows a unit test for parsing PropositionalWorld objects from JSON.

```
describe("parsing json", () => {
  describe("propositional worlds", () => {
    const signature: PropositionalSignature = new Set(["a", "b", "c"]);
    const worldParser = new PropositionalWorldParserFactory(signature);

    test.each([
      [`["a"]`, new PropositionalWorld(signature, new Set(["a"]))],
      [`["a", "b"]`, new PropositionalWorld(signature, new Set(["a", "b
          "]))],
      [`["c", "a", "b"]`, new PropositionalWorld(signature, new Set(["a
          ", "b", "c"]))],
      [`[]`, new PropositionalWorld(signature, new Set([]))],
    ])("parse: %j", (input: string, expected: PropositionalWorld) => {
      expect(worldParser.fromJson(input).assignment).toEqual(expected.
          assignment);
      expect(worldParser.fromJson(input).signature).toEqual(expected.
          signature);
    });
  });
});
```

Listing 9: Excerpts from a unit test (`src/test/parse-json.test.ts`)

Example output when running all tests suites is shown in Listing 10. As visible there, the continuous integration server also displays what lines of the code are not covered with tests. For future work, the test coverage metrics can be integrated into the checks for every commit, ensuring only tested code is introduced.

```
PASS src/test/parse-json.test.ts
PASS src/test/serialize-json.test.ts
PASS src/test/serialize-binary.test.ts
PASS src/test/parse-binary.test.ts
-------------------------------------|---------|----------|---------|---------|--------------------
File                                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------------------------|---------|----------|---------|---------|--------------------
All files                            |    81.2 |    59.62 |      75 |   79.82 |
 logic                               |    78.1 |    56.52 |      75 |   75.53 |
  BeliefRevisionInput.ts             |     100 |      100 |     100 |     100 |
  DMFSystem.ts                       |   21.74 |        0 |   33.33 |   18.18 | 22-62
  DMFSystemState.ts                  |   83.33 |      100 |   66.67 |      75 | 9
  PropositionalWorld.ts              |     100 |      100 |     100 |     100 |
  WorldPreference.ts                 |   93.22 |    84.62 |     100 |   92.59 | 84-88
 serialize                           |    88.7 |    62.07 |     100 |   88.18 |
  BeliefRevisionInputParserFactory.ts |     100 |      100 |     100 |     100 |
  DMFSystemParserFactory.ts          |     100 |      100 |     100 |     100 |
  DMFSystemStateParserFactory.ts     |     100 |      100 |     100 |     100 |
  PropositionalWorldParserFactory.ts |      70 |       40 |     100 |      70 | 39-43,51-55
  WorldPreferenceParserFactory.ts    |   90.14 |    73.68 |     100 |   89.39 | 18,45,60-64,111
 util                                |   42.86 |      100 |       0 |   42.86 |
  functions.ts                       |   42.86 |      100 |       0 |   42.86 | 7-11,21-22,27,31-38
-------------------------------------|---------|----------|---------|---------|--------------------

Test Suites: 1 skipped, 4 passed, 4 of 5 total
Tests:       1 skipped, 52 passed, 53 total
Snapshots:   0 total
Time:        4.551 s
Ran all test suites.
```

Listing 10: Running all test suites in logic-ts

logic-components

logic-components [Hel21a] is a library of web components that provide commonly used elements for tools working with logic. It enables users to embed signatures, propositional worlds or total preorders (that can be changed by drag & drop), and more. Figure 26 shows an example for a total preorder as rendered by logic-components.

Recent years have seen a rapid maturing of the JavaScript ecosystem with the introduction of single-page-application frameworks like Angular[8], React[9] and Vue.js[10]. A downside of the fast rate of change is framework-churn, meaning hard to maintain projects because the underlying framework had too many breaking changes. To avoid these problems logic-components is implemented using *Custom Elements*[11]. Custom elements are part of the HTML standard and are widely encouraged by the industry (e.g., by Git-

[8] `https://angular.io/`
[9] `https://reactjs.org/`
[10] `https://vuejs.org/`
[11] `https://html.spec.whatwg.org/multipage/custom-elements.html`

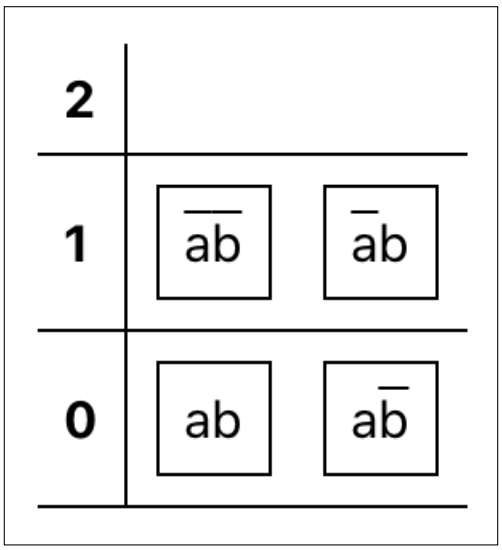


Figure 26: Total preorder over $\Sigma = \{a, b\}$ from logic-components

Hub[12] or Google[13]). In combination with other standard APIs like the shadow dom and HTML templates, custom elements allow developers to create reusable web components that run natively in modern browsers. Various toolchains exist to make the development of web components easier, here Stencil[14] was used to built logic-components. In addition to providing an easier-to-use API, Stencil also automatically generates documentation about the properties and dependencies of all web components that can be found directly in the repository. Most importantly, the generated web components are independent of any framework.

While web components can be used with JavaScript only, their integration into a framework can be challenging. Because of problems with change detection, to use logic-components with the popular React framework, every component needs to be wrapped into a custom React component. Consequently, this makes using logic-components with React error-prone and forces developers to write boilerplate code. An additional compilation step was implemented that automatically creates these wrapper components and publishes them as an npm package (logic-components-react[15]) to solve these problems.

[12] `https://github.blog/2021-05-04-how-we-use-web-components-at-github/`
[13] `https://developers.google.com/web/fundamentals/web-components/customelements`
[14] `https://stenciljs.com/`
[15] `https://www.npmjs.com/package/@rhazn/logic-components-react`

dmf-revision

The web application that combines logic-ts and components from logic-components is implemented in the *dmf-revision* package. A large part of the needed functionality is related to user interaction and display of the underlying data. For that reason, React was chosen as a frontend framework instead of relying on plain TypeScript. React is a popular library for writing user interfaces, developed by Facebook. It was chosen with the FAIR principles in mind as their design principles[16] include interoperability and stability.

Reacts structure favors splitting the functionality into smaller parts, called components. Building larger elements of the user interfaces is then done by combining components using composition. This closely aligns with the well-known principle of favoring object composition over inheritance [GHJV95] and allows reuse of components. The export of configuration for DMF revision, shown in Figure 27, is an example of this principle. In this screenshot, every button is a component that contains shared logic for rendering, displaying a download dialog, and saving a file. Because the common logic is already implemented in the component, the page that allows the user to export DMF configuration only has to provide the content.

An example is `DMFEdgeDisplay`, a component that renders an edge in a DMF system. Listing 11 shows excerpts of the code. It is a functional component that takes a list of parameters as input and returns data that can be rendered by React. Coeus makes extensive use of dependency injection for services. On line 2 an instance of `DMFService` is provided to the component from the context. This service uses the postulate-web-server API to transform formulas from TPTP syntax to TeX for display. During development, the object is instantiated to `localhost`. For production deployments, it is exchanged for the production URL of the API. The components manage their state using React hooks. The `useState` hook returns a variable that references the current value and a function to set a new value. Line 14 shows how this functionality is used in a second hook, `useEffect`. The useEffect hook is executed every time its dependencies change (here index or hideName) and sets a new name for the edge display. Finally, in line 18, the component returns data that is rendered by React. Every time the state of the component changes (e.g., in the useEffect hook described previously), this data is updated with its new value and re-rendered.

The component-based approach also naturally extends to web components. The user interface elements that are implemented in logic-components are embedded into pages

[16] `https://reactjs.org/docs/design-principles.html`

```tsx
export const DMFEdgeDisplay: React.FC<DMFEdgeDisplayProps> = ({ edge, index, hideName }: DMFEdgeDisplayProps) => {
    const dmfService = useContext(DMFServiceContext);
    const [formulaLatex, setFormulaLatex] = useState<string>("");
    const [name, setName] = useState<string>("");

    useEffect(() => {
        (async () => {
            const formulaLatex = await dmfService.getFormulaLatex(edge.formula);

            setFormulaLatex(formulaLatex);
        })();
    }, [edge, dmfService]);

    useEffect(() => {
        setName(!!hideName ? `` : String.raw`e_{${index}}\textrm{: }`);
    }, [index, hideName]);

    return (
        <div>
            <MathComponent
                display={false}
                tex={String.raw`${name}(${contextTpoNameByIndex(
                    edge.fromIndex,
                )}, ${formulaLatex}, ${contextTpoNameByIndex(edge.toIndex)})`}
            />
        </div>
    );
};
```

Listing 11: Excerpts from DMFEdgeDisplay, displaying an edge of a DMF system in Coeus (`src/components/dmf-revision/DMFEdgeDisplay.tsx`)

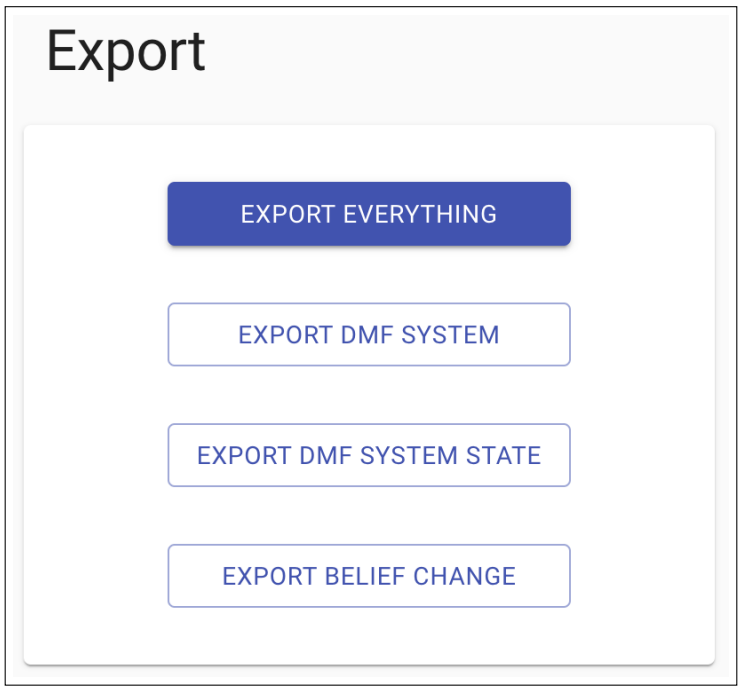


Figure 27: Export of DMF revision configuration. Every button is a component that includes shared implementations for opening a download dialog and saving a file to disk. Different configuration leads the components to save different content as well as be rendered differently.

the same way as components from dmf-revision.

In addition to functional components, functional programming is also used for small utilities because functions without side effects are easy to reason about and test. An example is shown in Listing 12. Here, the additional syntax needed for a well-formed TPTP first-order formula is added to a formula.

```
1 export const addTptpBoilerplate = (formula: Formula): string => {
2     return `fof('formula', axiom, ${formula}).`;
3 };
```

Listing 12: Example utility function, adding TPTP syntax for a first-order formula to a raw formula input. (`src/util/functions.ts`)

Services were implemented as an abstraction layer over the API and allow components to call the HTTP endpoints of postulate-web-server asynchronously. As described previ-

ously, dependency injection using React context is used to provide a service object with the correct API base URL to components. The services themselves also use dependency injection. In Listing 13, line 2 shows constructor injection to get a correctly configured instance of `SyntaxCheckService`. The function `checkPropositional`, defined on line 4, is implemented using the JavaScript async/await API to execute non-blocking. It sends a formula in TPTP syntax to postulate-web-server to check whether or not it is a valid, propositional formula and then parses the response from JSON and returns it.

```
1  export class SyntaxCheckService {
2      constructor(private host: string) {}
3
4      public async checkPropositional(formula: Formula): Promise<
           SyntaxCheckResponse> {
5          return fetch(`${this.host}/syntax/tptp/propositional`, {
6              method: "PUT",
7              headers: {
8                  "Content-Type": "application/json",
9              },
10             body: JSON.stringify({
11                 formula: formula,
12             }),
13         }).then(response => response.json());
14     }
15 }
```

Listing 13: Excerpts from SyntaxCheckService that provides functionality to check if a TPTP formula is valid. (`src/service/SyntaxCheckService.ts`)

A small selection of open source projects was used to implement the frontend of Coeus in the limited time.

Formulas and mathematical symbols are rendered with MathJax[17] using React components from mathjax-react[18]. As described in Subsection 5.4, TPTP syntax is still used for formula input but MathJax allows Coeus to display most elements in the familiar mathematical notation.

[17] `https://www.mathjax.org/`
[18] `https://github.com/charliemcvicker/mathjax-react`

Because DMF systems can be thought of as directed graphs, D3.js[19] and react-d3-graph[20] are used to render them as such. The representation of a DMF system as a directed graph allows users to quickly get an overview of how an agent will switch between the total preorders, which is especially valuable when configuring belief revision inputs.

Finally, Material-UI[21], a popular component library for React was used as a basis to create the user interface.

During the implementation, the user interface was changed multiple times with feedback. As a prototype, in a direct extension of previous work, all information was presented on one page. This approach led to overwhelming complexity as anticipated in Subsection 4.4. To guide the user, the UI was broken up into multiple steps. Figure 28 shows the horizontal stepper on top and how only information about the signature is shown. Every individual page only displays part of the configuration but still provides the user with their overall context in the process.

The downside of organizing the process into multiple steps was that users were now missing context. Especially during configuration of belief revision input, common feedback was that the derived-faithful tpos $\leq_{\Psi}^{M}$ of the DMF state Ψ as well as the DMF system M would be helpful. An overlay system was implemented (see Figure 29) to enable users to reference these elements without overloading the page. With this approach, only the most important information is shown. Experienced users can still reference other elements of the configuration if needed.

5.8. Implementation of DMF revision

When a user enters a new belief change input α in Coeus the posterior DMF revision state $s_{\Psi\circ\alpha}^{M}$ is automatically computed. In contrast, previous work relied on users entering the posterior epistemic state by hand. A DMF revision operator for epistemic states, $\circ$, therefore needed to be implemented.

A derived-faithful tpo $\leq_{s_{\Psi}^{M}}$ to a DMF system state s_{Ψ}^{M}, as described in Subsection 4.3, was implemented in postulate-web-server. The Java source code can be seen in Listing 14.

[19] `https://d3js.org/`
[20] `https://github.com/danielcaldas/react-d3-graph`
[21] `https://material-ui.com/`

```java
public TotalPreorderState getDerivedFaithfulTpo(TotalPreorderState
    contextTpo,
                        List<Formula<PropositionalSignature<Character
                            >>> beliefSet) {
  PropositionalLogic<Character> logic = new PropositionalLogic<>();

  Set<PropositionalInterpretation<Character>> beliefSetModels =
      contextTpo.getLayers().stream()
       .flatMap(
           layer -> layer
               .stream()
               .filter(
                   interpretation -> beliefSet.stream()
                       .allMatch(formula -> logic.satisfies(
                           interpretation, formula)))
       ).collect(Collectors.toUnmodifiableSet());

  return getDerivedFaithfulTpo(contextTpo, beliefSetModels);
}

public TotalPreorderState getDerivedFaithfulTpo(TotalPreorderState
    contextTpo,
                        Set<PropositionalInterpretation<Character>>
                            beliefSetModels) {
  List<Set<PropositionalInterpretation<Character>>> layers = contextTpo
      .getLayers().stream().map(
       layer -> layer
           .stream()
           .filter(
               Predicate.not(beliefSetModels::contains))
           .collect(Collectors.toUnmodifiableSet())
  ).collect(Collectors.toList());

  layers.add(0, beliefSetModels);

  return new TotalPreorderState(layers, contextTpo.getSignature());
}
```

Listing 14: Constructing a derived-faithful tpo from a DMF system state (`src/main/java/com/rhazn/service/DMFRevisionService.java`)

With the ability to compute derived-faithful tpos to DMF system states, a DMF revision operator according to Definition 11 can be implemented.

The belief set of the posterior DMF system state is constructed on the server by first selecting the minimal models of the input α in $\leq_{s_\Psi^M}$. For a minimal formula γ so that $[\![\gamma]\!] = min([\![\alpha]\!], \leq_{s_\Psi^M})$, initially, the full disjunctive normal form is computed from the set of minimal models. The resulting formula is then minimized using the Quine–McCluskey-Algorithm [McC56] and returned to the frontend.

Because the definition of the DMF system M is managed by the frontend, finding $\preceq_{\Psi \circ \alpha}$ is implemented there. As candidates for a context-switch, the frontend requests a list of all outgoing edges from the current context tpo $\preceq_\Psi$ of the current DMF system state. Their associated formulas are sent to the backend and checked for semantic equivalence with the input α. If a semantically equivalent formula is found, the context tpo of its associated edge is set as the next context tpo. If not, no change is made.

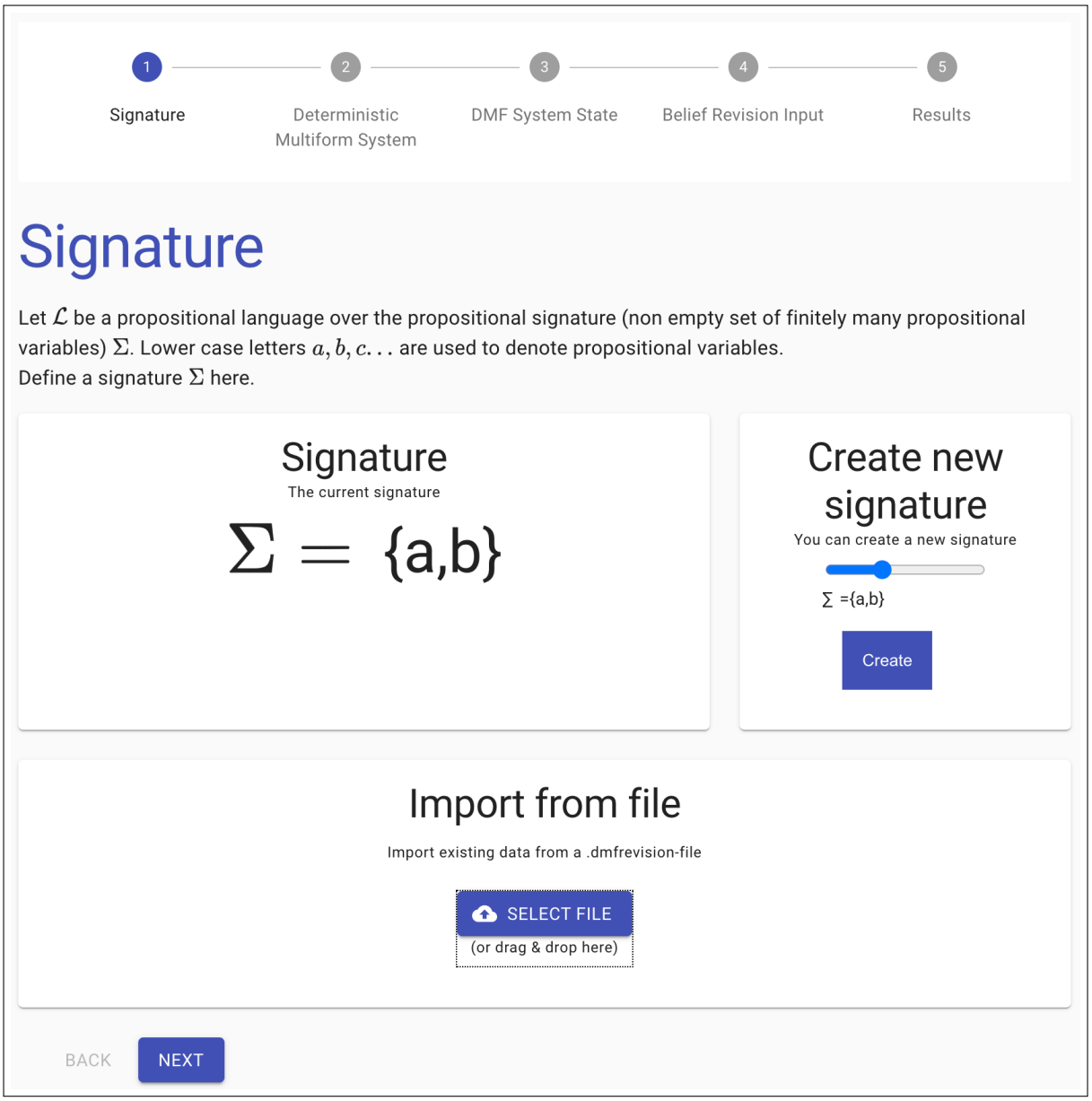


Figure 28: The user interface of Coeus is broken up into multiple pages using a horizontal stepper

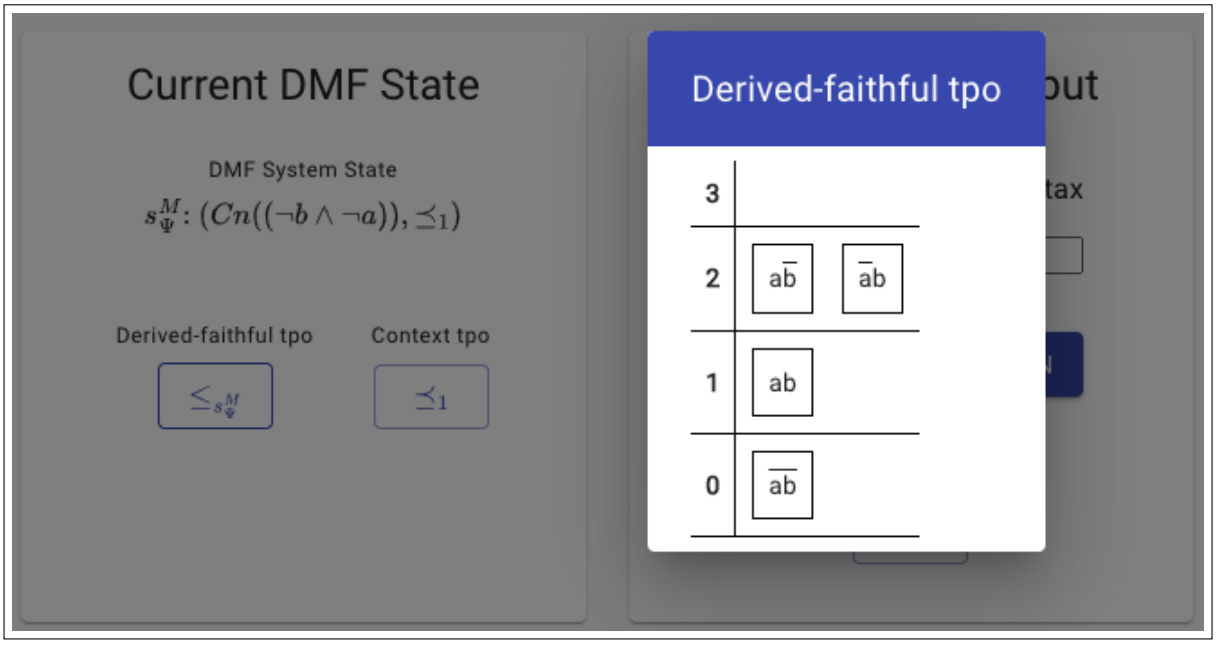


Figure 29: Overlays are used to enable users to view more information as needed, here of a derived-faithful tpo $\leq^M_\Psi$

6. Evaluation & Improvements

For an evaluation of the implementation, recall the goals for Coeus, outlined in Subsection 4.1. The application aims to support experimental studies on DMF revision by researchers and can also have applications for didactic purposes. It should be able to automatically verify whether or not an iterated DMF revision satisfies a set of postulates from literature. The configuration should be shareable and the tool easy to distribute and install. To support future work, a set of guidelines, called the FAIR principles of research software, should be followed. They promote the usage of good software development practices and adherence to established standards.

The anticipated challenges for successful implementation were discussed in Subsection 5.1. They could largely be divided into technical challenges, decisions about standards for textual representation, and user interface questions. A similar structure will be followed in this chapter by first discussing implementation choices and improvements before moving on to usability feedback.

Implementation choices

The biggest decision regarding software architecture was building Coeus as a browser-based Client/Server application. This was mainly done to support the easy distribution of the tool. During the writing of this work, the architecture already proved to be helpful when gathering feedback. It also provided a quick way to distribute updates when making frequent changes. A downside of the architecture is the added overhead of having to implement a server to communicate with the existing logic library. While this increased the complexity of the project, the impact was limited because communication with the text-based HTTP protocol could reuse the textual representations already implemented for import and export functionality.

On the client-side, relying on TypeScript as a type-safe alternative to JavaScript paid of by catching many bugs already at compile time. Moreover, the ability of modern IDEs to provide type hints for TypeScript projects was of particular help during the development of a complex application. Similarly, the choice of StencilJS as a toolchain to write web components provided good documentation and reduced the complexity of the web components API. While StencilJS enabled an automated generation of components to be used with the React framework, in hindsight, the incompatibilities were severe enough to introduce risk to the implementation. Building web components as a standard compliant basis for a web application is promising but still has interoperability issues.

The server required Java to extend the existing implementation. The logic library was useful and flexible. Because the backend is written in Java, it was also possible to use other JVM-based projects like the scala-tptp-parser [Ste21] that sped up development. Because back- and frontend are implemented in different technology stacks, the software is quite complex. Parts of it (e.g., parsing the textual representation) needed to be implemented twice. This could have been avoided by either using a language like NodeJS in the backend or implementing the client as a Java program.

Overall the implementation choices for Coeus were reasonable and provided a solid basis for a modularized and reusable implementation. Less complex choices would have been possible by sacrificing some of the desirable properties from the FAIR principles.

Software quality

Outlining the desirable properties of the implementation at the start was a helpful guideline during the work. Developing research software requires additional considerations to industry software, mainly the ability to find and cite specific versions of software artifacts. Hosting the code on GitHub and distributing JavaScript modules with npm meant that the implementation was easy to find and share. Using Zenodo to make concrete versions citeable was easy to set up and worked well.

A potential problem with presenting the software in so many different ways (GitHub for collaborative work on source code, npm to use the compiled JavaScript and Zenodo to cite) was the amount of work that would need to be done for every release. Implementing automated build and release scripts was therefore crucial. GitHub actions were easy to implement and provided the necessary features. Coupled with the well-designed auto-import of repositories into Zenodo, the complexity was manageable.

In regards to good software development practices and automated builds, automated testing plays an important role. In that regard, mainly automated unit tests were implemented. These unit tests allowed for fast iterations during the development of textual representations in logic-ts and parsing of TPTP formulas in postulate-web-server. Automatically verifying the correctness of the tested code when making changes caught multiple bugs. For other parts of Coeus, especially those that provide a user interface like dmf-revision, end-to-end testing (testing the complete functionality) would have been appropriate. Due to the complexity of writing end-to-end tests as well as frequent changes in application flow during development, no such tests have been implemented. As a result, some parts of the application do not meet the goal of being tested automatically and should be improved as soon as possible.

The FAIR principles include interoperability and relying on existing standards. For the textual representation, adhering to standards paid off. Especially using TPTP syntax consistently to work with formulas had multiple benefits. Most importantly, because the syntax is used in automated theorem proving, a variety of parsers already exist. The scala-tptp-parser could therefore be used instead of implementing a parser. Moreover, TPTP allowed for an implementation of propositional and first-order logic formulas with only minor changes. Using JSON to represent tpos instead of a custom binary representation was a second existing standard that was used. During the development of Coeus, it was helpful to be able to read the textual representation of a tpo (e.g., in an HTTP request). The fact that JSON is easy to manipulate in multiple programming languages was also important because parsing the representation needed to be implemented twice (once in Java and once in TypeScript).

Good modularization was the main benefit of following good software architecture. During the creation of this work, it was possible to create multiple other tools (Alchourron [SH21, SHB21] as well as Preference Builder [Hel21c]) because the generic modules could be reused. In addition to allowing the rapid implementation of these tools, this also meant that future improvements and bug fixes were available to all applications using the modules.

Model-checking approach

Verification of postulates for belief change was done by employing model-checking. This provided a list of benefits: Model-checking is conceptually easy to understand, which made the extension of an existing approach possible and will make Coeus a better didactic tool. The implementation is fully automated and easy to extend with new postulates, an important quality for research software. With future work, it will be possible to provide counter-examples for postulates that are not satisfied.

Despite these benefits, the model-checking approach also has drawbacks, most importantly the limited scalability with input size. Specifically, additional work (see Subsection 6.1) was needed to allow the initial implementation to work well with a signature of size three. Very likely, the current prototypical functionality of evaluating quantified formulas for all elements of the universe can be improved further, but that was considered outside the scope of this work. Mitigating the performance limitations of model checking is the fact that belief revisions are configured by hand. Because tpos as a representation of agent state also expand quickly with signature size, belief revisions created by users will be relatively small.

It would be interesting to extend the approach from verifying individual belief revisions to verifying postulates for a complete operator. For that purpose, the current implementation would need to be extended, possibly moving away from model-checking to automated theorem proving. Again this was considered to be out of scope for this prototypical implementation.

6.1. Performance improvements

A summary of the work done in the scope of this thesis is included in the paper by Kai Sauerwald and the author [SH21]. This chapter presents an extended report and more details on the completed performance improvements.

Model-checking is an inherently complex problem. While a change of the core algorithms was out of scope for this work, a survey of performance problems was taken and major problems addressed.

All performance measurements were taken with the docker build of postulate-webserver (as described in Section B) running with resources of 4 CPUs and 8GB of ram on a MacBook Pro (16-inch, 2019). The target workload was the `/postulateCheck` route that checks whether a belief change satisfies any of the preconfigured postulates. All measurements were run four times and averaged. The input belief change always consisted of the formula a, an initial tpo with all worlds considered equal, and a posterior tpo with all a-worlds in the lowest rank to create the maximum amount of computation needed. The input for the signature of size three is shown in Listing 15, smaller signatures followed the same structure. While the absolute numbers depend on the system, overall trends could be identified.

```
{
    "formula": "fof('formula', axiom, a).",
    "signature": ["a","b","c"],
    "beforeTPO": {"signature":["a","b","c"],"ranks":[[["a","b","c"],["b
        ","c"],["a","c"],["c"],["a","b"],["b"],["a"],[]]]},
    "afterTPO": {"signature":["a","b","c"],"ranks":[[["a","b","c"],["a
        ","c"],["a","b"],["a"]],[["b","c"],["c"],["b"],[]]]}
}
```

Listing 15: Example belief change input

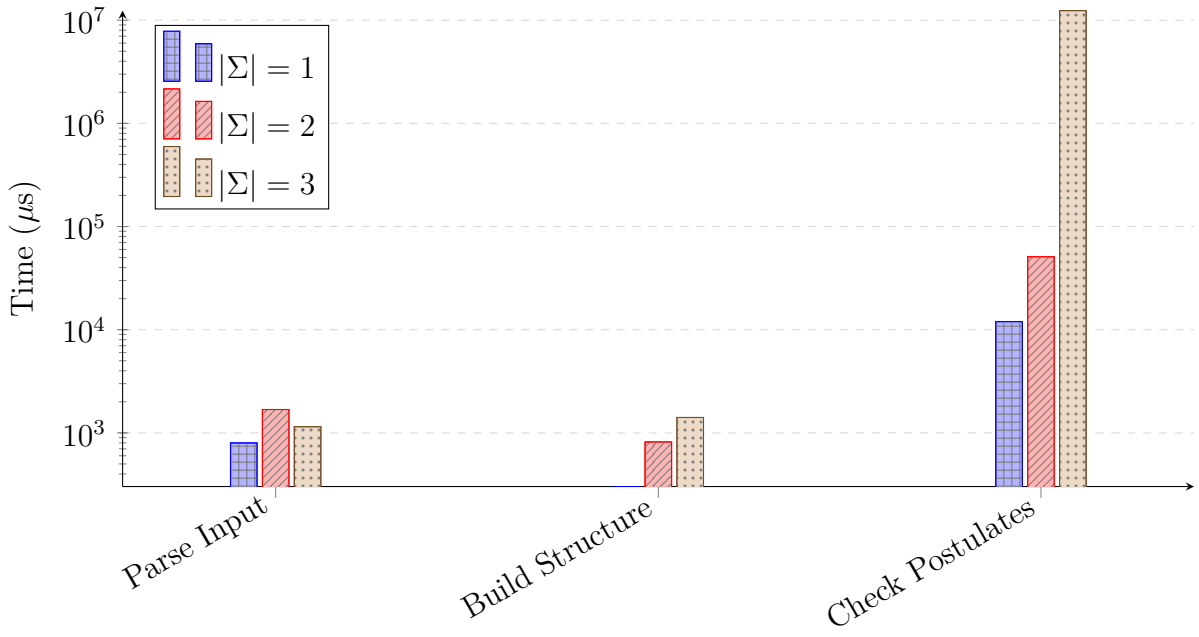


Figure 30: Scaling of response time with signature size (log)

Initially, performance scaling of the main stages for the algorithm was measured. Postulate-check-server performs the following main functions to handle one HTTP request:

1. Parse Input: Parse the request body JSON and build a belief change structure from
2. Build Structure: Build the structure $\mathcal{A}_\mathcal{C}$ (see Subsection 4.2)
3. Check postulates: For all postulates, certify if they satisfy the input belief change or not

Measurements were taken in nanoseconds using the Java `Instant.now()` and `Duration.between()` APIs. Figure 30 shows the results for signature sizes from one to three elements, plotted in log scale. From the results, it is clear that parsing the request as well as building the initial structure show relatively little growth. The most important target for optimization would be the verification of postulates.

During the performance improvements, three main changes were applied:

1. Using GraalVM [22] instead of OpenJDK to run postulate-check-server

[22] `https://www.graalvm.org/`

Change	**Average time**	**Speedup** (Initial Time / New Time)
Initial	12.5s	1
GraalVM	15.3s	0.82
Parallel Postulates	7.1s	1.76
Parallel Quantifiers	3.9s	3.21

Table 1: Response time changes

2. Making sure postulates are evaluated in parallel

3. Running evaluation of quantified formulas in parallel

The results, as well as the speedup factor to the initial state, are shown in Table 1. A request with a belief change over a signature of size three took 12.5 seconds on average initially and could be reduced to 3.9 seconds by the end.

GraalVM is an alternate runtime for, besides others, Java applications. One use case is as a drop-in replacement to improve performance by employing additional optimizations as described by Stadler et al. [SWM14]. Performance improvements with GraalVM are well documented by industry (such as Twitter[23] or Facebook[24]) and the set up promises to be relatively easy.

At the time of writing, GraalVM was available for Java 8 and Java 11 with an experimental version for Java 16. Because logical-systems (and by extension postulate-web-server) requires Java 15, only the experimental GraalVM version for Java 16 could be tried. GraalVM provides Docker images to run applications but only provides those for Java 8 and Java 11, the experimental GraalVM version for Java 16 was not yet available. To test GraalVM, therefore the experimental GraalVM JDK for Java 16 was downloaded and used to build postulate-web-server. The local Dockerfile was changed to be identical to the Docker image that builds the GraalVM image for Java 11, replacing only the `JAVA_VERSION` argument with `java16`.

Contrary to expectations, using GraalVM to run postulate-check-server decreased performance as shown in Table 1. Because this result was unexpected, the performance measurements with GraalVM were repeated multiple times with similar outcomes. Reverting the changes to the Dockerfile and just running the JAR, built with the GraalVM compiler and using the OpenJDK VM, also showed degraded performance. A possible explanation would be a mistake with either the GraalVM compiler or creating the

[23] `https://www.youtube.com/watch?v=pR5NDkIZBOA`
[24] `https://medium.com/graalvm/graalvm-at-facebook-af09338ac519`

GraalVM Java 16 Docker image. It also seems plausible that the experimental nature of the GraalVM for Java 16 could be the reason for the missing performance gains. The test should be repeated once it reaches a stable version. Any GraalVM related change was reverted before continuing.

For the following optimizations, special attention was paid to increasing the amount of parallelism. During initial performance measurements, every individual postulate check was timed. Most time was spend checking (CR5) and (CR6) with about six seconds each (see Figure 31. Even though the postulate check was implemented using `parallelStream()` on a `HashMap` of postulates, (CR5) would consistently be checked only after (CR6) was completed.

Because splitting work over multiple threads and rejoining it at the end carries an overhead, the Java Stream implementation does not necessarily split tasks into individual elements. Java provides so-called spliterators to decide how to split a parallel workflow. Array-based spliterators split elements evenly until a minimum size is reached. Because the internal implementation of `HashMap` is always initialized with 16 elements (and new elements are placed depending on their hash), a `HashMap` with few elements is a sparse array that might have an unbalanced distribution of elements. Consequently, multiple expensive operations might end up in the same split and be distributed to the same thread. If the processing costs for an individual element can be high, the `HashMap` can be copied in an array with a known size (like an `ArrayList`) during a preprocessing step. Doing so enables the spliterator to create balanced splits. Implementation of such a preprocessing step allowed all postulates to be processed in parallel, nearly halving the response time of postulate-web-server from 12.5 seconds to 7.1s as seen in Table 1.

In a final optimization, the evaluation of formulas in logical-systems was considered. The reason for the certification of (CR5) and (CR6) scaling worse than other postulates was that they are the only postulates that include three all quantifiers. In logical-systems, quantifiers are implemented using the `allMatch()` and `anyMatch()` functions on all elements of the universe of an interpretation. This work could also be parallelized using `parallelStream()`. The time needed to check individual postulates is shown in Figure 31. With improved formula evaluation in logical-systems, it was possible to improve the processing time of postulates with multiple quantifiers massively. In combination with the improvements to the parallel evaluation of postulates, this led to a response time decrease from initially 12.5 seconds to finally 3.9 seconds (Table 1) for a belief change over a signature with three elements.

The performance improvements done focussed on improving the concrete implement-

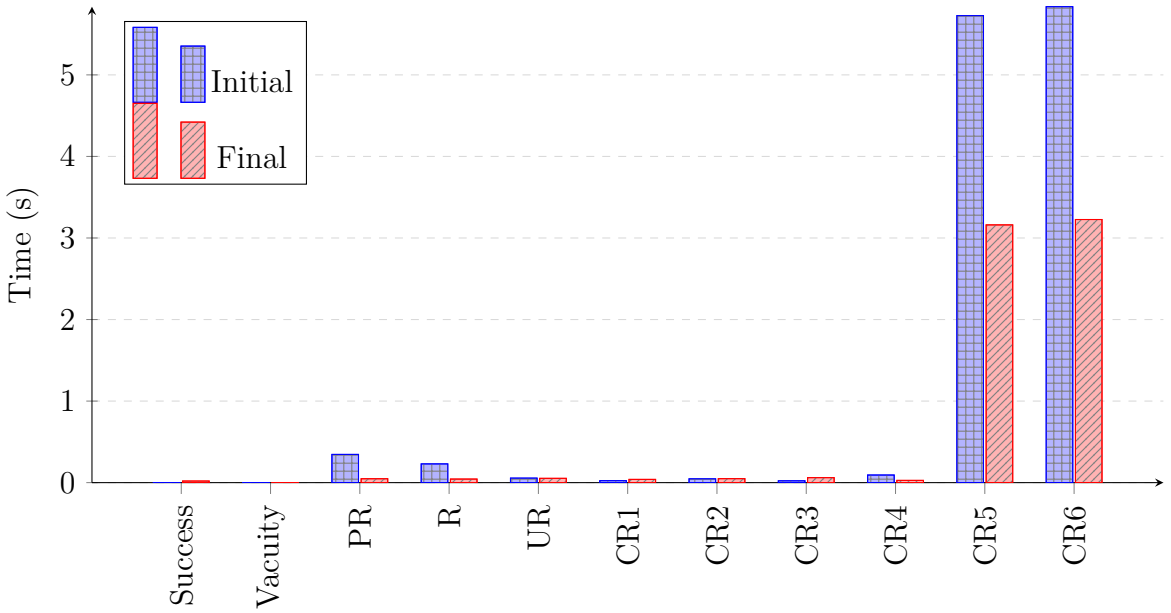


Figure 31: Individual postulates, $|\Sigma| = 3$

ation of the model-checking approach. Because Coeus is designed as a tool that requires users to create belief change inputs by hand, it can be reasonably assumed that signature sizes will be small. While response times for that use case are satisfactory, the approach inherently scales poorly with signature size. Future work should focus on improving the actual core algorithm to enable logical-systems to handle larger signature sizes.

Usability

As previously identified, the amount of information that has to be entered for DMF revision created a usability challenge. In a first iteration, Coeus naively extended previous work done with WHIWAP [SH19] and presented all data on a single page. During testing, this quickly felt hard to understand, and the input was moved into multiple parts and guided by a horizontal stepper. Showing only parts of the configuration makes it harder for users to get an overview of input but allows for more focus on the currently important information. While the tool is still complex, splitting the interface up improved the user experience significantly.

The advisor, as well as members of the chair of knowledge-based systems, were asked to review earlier versions of the implementation to gather feedback from researchers. Approaching potential users proved invaluable and led to many changes, improving usability and information display.

Configuring a belief revision operator based on human input imposes limitations on its complexity. In particular, tpos grow with $|\Omega|$ (and therefore signature size). Because users have to create tpos as part of the configuration, an operator over a large signature or many tpos is hard to define. For the implementation of Coeus, this tradeoff was considered acceptable (especially because the model checking approach is also limited by performance). In the context of didactic and small experiments, the visibility of tpos can even be considered a strength. Still, the manual input presents a limitation of the software. Future work could extend it to a more programmatic configuration of operators.

7. Conclusion

Summary

In summary, this work presented an approach to iterated revision based on deterministic, multiform systems as well as an implementation.

Section 1 and Section 2 introduced necessary notation as well as prior work. Seminal work in belief revision like the AGM postulates [AGM85] and the epistemic state framework [DP97] was presented. Prior work concludes with more recent work on uniform revision [AB01, Ara20].

Extending uniform revision by deterministically switching between total preorders is the core concept of deterministic multiform revision. As a theoretical foundation for the following implementation, Section 3 defined what a deterministic multiform system is and how it is applied to an extended agent state. Iterated belief revision in the DMF framework was introduced.

A web application for experimental studies of DMF revision, called Coeus, was implemented as part of this work. A more theoretical introduction to research software and the implemented algorithms was presented in Section 4. Implementation and architecture decisions were then shown in Section 5, followed by a more detailed overview of the individual modules of the software. Examples were highlighted in source code or figures to illustrate the functionality of the implementation. Challenges such as textual representations were pointed out, and solutions were discussed.

Finally, Section 6 provided a critical evaluation of the tradeoffs made during implementation and presented improvements to mitigate performance problems. Based on the initial goals and challenges, the section discussed decisions made and how they worked out during development. Limitations of the implementation were identified, and possible future extensions were presented.

Outlook

The introduction of DMF revision presents an opportunity to combine approaches from research in finite state machines and iterated belief revision. Supported by experiments using Coeus, researchers will be able to explore the properties of DMF revision operators.

Building on the model-checking approach to the verification of postulates, it will be a future challenge to extend the automated certification to more than a single belief change. Widening the input to whole operators will provide insights but also require

extending the underlying implementation.

Finally, the web components for building new applications for belief revision that were created during the course of this work will be extended and maintained as open-source.

References

[AB01] Carlos Areces and Verónica Becher. Iterable AGM functions. In *Applied Logic Series*, pages 261–277. Springer Netherlands, 2001.

[AGM85] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50(2):510–530, June 1985.

[Ara20] Theofanis Aravanis. On uniform belief revision. *Journal of Logic and Computation*, 30(7):1357–1376, September 2020.

[Bec99] Kent L. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[BHV16] Hudson Borges, André C. Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 334–344. IEEE Computer Society, 2016.

[BM06] Richard Booth and Thomas Meyer. Admissible and restrained revision. *Journal of Artificial Intelligence Research*, 26:127–151, June 2006.

[BM11] Richard Booth and Thomas Meyer. How to revise a total preorder. *Journal of Philosophical Logic*, 40(2):193–238, February 2011.

[Boo02] Richard Booth. On the logic of iterated non-prioritised revision. In *International Workshop on Conditionals, Information, and Inference*, pages 86–107. Springer, 2002.

[DP97] Adnan Darwiche and Judea Pearl. On the logic of iterated belief revision. *Artificial Intelligence*, 89(1-2):1–29, January 1997.

[EO13] European Organization For Nuclear Research and OpenAIRE. Zenodo, 2013.

[FH11] Eduardo Fermé and Sven Ove Hansson. Agm 25 years: Twenty-five years of research in belief change article in journal of philosophical logic. *Journal of Philosophical Logic*, 40:295–331, 2011.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://www.martinfowler.com/articles/injection.html`, January 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[Gä84] Peter Gärdenfors. Epistemic importance and minimal changes of belief. *Australasian Journal of Philosophy*, 62(2):136–157, June 1984.

[HCH+20] Wilhelm Hasselbring, Leslie Carr, Simon Hettrick, Heather Packer, and Thanassis Tiropanis. From FAIR research data toward FAIR and open research software. *it - Information Technology*, 62(1):39–47, February 2020.

[Hel21a] Philip Heltweg. rhazn/logic-components. `https://doi.org/10.5281/zenodo.4742731`, 2021.

[Hel21b] Philip Heltweg. rhazn/logic-ts. `https://doi.org/10.5281/zenodo.5078529`, 2021.

[Hel21c] Philip Heltweg. rhazn/preference-builder. `https://doi.org/10.5281/zenodo.5078533`, 2021.

[JT07] Yi Jin and Michael Thielscher. Iterated belief revision, revised. *Artificial Intelligence*, 171(1):1–18, 2007.

[KM91] Hirofumi Katsuno and Alberto O. Mendelzon. Propositional knowledge base revision and minimal change. *Artificial Intelligence*, 52(3):263–294, December 1991.

[Kut19] Steven Kutsch. Infocf-lib: A java library for ocf-based conditional inference. In Christoph Beierle, Marco Ragni, Frieder Stolzenburg, and Matthias Thimm, editors, *Proceedings of the 8th Workshop on Dynamics of Knowledge and Belief (DKB-2019) and the 7th Workshop KI & Kognition (KIK-2019) co-located with 44nd German Conference on Artificial Intelligence (KI 2019), Kassel, Germany, September 23, 2019*, volume 2445 of *CEUR Workshop Proceedings*, pages 47–58. CEUR-WS.org, 2019.

[McC56] Edward J McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[NPP03] Abhaya C. Nayak, Maurice Pagnucco, and Pavlos Peppas. Dynamic belief revision operators. *Artificial Intelligence*, 146(2):193–228, 2003.

[Pep13] Pavlos Peppas. A panorama of iterated revision. In *David Makinson on Classical Methods for Non-Classical Problems*, pages 71–94. Springer Netherlands, December 2013.

[Sau21] Kai Sauerwald. Logical Systems. `https://github.com/Landarzar/logical-systems`, 2021.

[SB21] Kai Sauerwald and Christoph Beierle. Multiform revision. (unpublished), 2021.

[SH19] Kai Sauerwald and Jonas Haldimann. Whiwap: Checking iterative belief changes. In *DKB/KIK@ KI*, pages 14–23, 2019.

[SH21] Kai Sauerwald and Philip Heltweg. On using model checking for the certification of iterated belief changes. In Christoph Beierle, Marco Ragni, Frieder Stolzenburg, and Matthias Thimm, editors, *Proceedings of the 7th Workshop on Formal and Cognitive Reasoning co-located with the 44th German Conference on Artificial Intelligence (KI 2021), Berlin, Germany, September 28, 2021*, volume 2961 of *CEUR Workshop Proceedings*, pages 23–33. CEUR-WS.org, 2021.

[SHB21] Kai Sauerwald, Philip Heltweg, and Christoph Beierle. Certification of iterated belief changes via model checking and its implementation. In Leila Amgoud and Richard Booth, editors, *19th International Workshop on Non-Monotonic Reasoning (NMR 2021), Hanoi, Vietnam, November 2-5, 2021, Proceedings*, 2021. (forthcoming).

[SM05] Friedrich Steimann and Philip Mayer. Patterns of interface-based programming. *The Journal of Object Technology*, 4(5):75, 2005.

[Spo88] Wolfgang Spohn. Ordinal conditional functions: A dynamic theory of epistemic states. In *Causation in decision, belief change, and statistics*, pages 105–134. Springer, 1988.

[Ste21] Alexander Steen. scala-tptp-parser. `https://doi.org/10.5281/zenodo.4672395`, April 2021.

[Sut17] Geoffrey Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[SWM14] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '14*. ACM Press, 2014.

[WDA+16] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016.

Appendices

A. Provided Files

File location	Note
3230880_philip_heltweg.pdf	Master Thesis PDF
master-thesis/	Master Thesis LateX source files
software/coeus/dmf-revision/	Frontend of Coeus as described in Subsection 5.7
software/coeus/logic-ts/	Logic library for TypeScript as described in Section 5.7
software/coeus/logic-components/	Web components for browser-based frontends as described in Section 5.7, [Hel21a]
software/coeus/logic-components-react/	React bindings for logic-components as described in Section 5.7
software/coeus/postulate-check-server/	Java API for logical-systems and DMF revision implementation as described in Subsection 5.6
software/logical-systems-parser/	Parser for formulas from TPTP syntax into logical-system objects as described in Subsection 5.5
software/whiwap2/	Extension of WHIWAP created from logic-components as described in [Sau21]
software/preference-builder	Tool created from logic-components to build preference-relations over worlds as described in Subsection 5.3

Table 2: Provided software

B. Setup and Deployment

Backend: postulate-check-server

postulate-check-server requires at least Java 15. If possible, dependencies are managed using Apache Maven [25]. Therefore, the project needs to be set up as a Maven project. `mvn install` can be executed to download dependencies.

Some private dependencies are not available in a maven repository. Because of that, the project has additional dependencies on the following modules locally:

[25] `https://maven.apache.org/`

1. enumerative-combinatorics
2. logical systems
3. alchourron
4. alchourron-postulates

Because not all packages are available publicly, a prebuilt JAR with dependencies is checked in at `/build/postulate-check-server.jar` to make running the project more accessible.

Deployment of both the back- and frontend is consistent as both are ultimately built as a docker image. The Dockerfile in the project root creates an image using OpenJDK [26] and expects a runnable JAR of the project at `/build/postulate-check-server.jar`. HTTPS can be enabled by providing both a certificate as a Java KeyStore file as well as its password using the environment variables `KEYSTORE_LOCATION` and `KEYSTORE_PASSWORD` respectively. If no certificate is available, the project runs without HTTPS.

In summary, to build and run postulate-check-server with HTTPS enabled the following steps are needed:

1. build a JAR with dependencies at `/build/postulate-check-server.jar`
2. place a `certificate.jks` in `/build`
3. run `docker-compose build . --tag "<tag>"`
4. run `docker run -d -p <external_port>:4567 -e KEYSTORE_LOCATION=/usr/app/certificate.jks -e KEYSTORE_PASSWORD='<password>' <tag>`

Frontend: dmf-revision

The Coeus frontend is an npm project and requires node version 14+ and npm 7+. Because all dependencies are managed by npm, all that is needed to install them is to run `npm install`. A local development server can be started by calling `npm run start`. To build a production application bundle, the command `npm run build --prod` is available. To configure where the API of a postulate-check-server instance is running an environment variable `REACT_APP_API_URL` can be set or provided in an `.env` file in the project root. Because Coeus follows the continuous integration guidelines outlined in Section 5.7 an up

[26] `https://openjdk.java.net/`

to date script that creates a production artifact is always available in `.github/workflows/ci.yaml`.

A webserver serving the frontend must rewrite any requests to non-existing files to index.html because it is a single-page application written in React. As a result, the project can be packaged as a docker image that contains a correctly configured Nginx, similar to *postulate-check-server*. Before creating a docker image, the location of the postulate-check-server API needs to be configured in the `Dockerfile` in the project root. To enable HTTPS, the files `/infrastructure/ssl/webserver.key` and `/infrastructure/ssl/webserver.pem` can be added to the project root and the `Dockerfile`.

To build and run the docker image execute the following:

1. `docker build . --tag=<tag>`
2. `docker run -d -p <external_port>:443 <tag>`

C. Implemented postulates

AGM Postulates Success and Vacuity

Postulates (AGM∗2) and (AGM∗4) [AGM85].

```
1 fof('Success',postulate, ! [W1] : ( ( int(W1) & mod(W1,op(E0,A)) ) =>
     mod(W1,A) ) ).
2 fof('Vacuity',postulate, ! [W1] : ( ( int(W1) & mod(W1,E0) & mod(W1,A)
     ) => mod(W1,op(E0,A)) ) ).
```

Listing 16: Success and Vacuity postulate in FO^{TPC} using TPTP syntax

PR

From [BM06] as well as [JT07] as independence postulate.

(PR) For $\omega_1 \in [\![\alpha]\!]$ and $\omega_2 \in [\![\neg\alpha]\!]$, if $\omega_1 \leq_\Psi \omega_2$ then $\omega_1 <_{\Psi * \alpha} \omega_2$

```
fof('PR', postulate,
    ! [W1,W2] : (
        (
            int(W1)
            & int(W2)
            & mod(W1, A)
            & ~ mod(W2, A)
            & lesseq(W1, W2, E0)
        )
        =>
        (
            strictless(W1, W2, op(E0, A))
        ) ) ).
```

Listing 17: PR postulate in FO^{TPC} using TPTP syntax

R

Recalcitrance postulate [Boo02, NPP03].

(R) For $\omega_1 \in [\![\alpha]\!]$ and $\omega_2 \in [\![\neg\alpha]\!]$, $\omega_1 <_{\Psi*\alpha} \omega_2$

```
fof('R', postulate,
    ! [W1,W2] : (
        (
            int(W1)
            & int(W2)
            & mod(W1, A)
            & ~ mod(W2, A)
        )
        =>
        (
            strictless(W1, W2, op(E0, A))
        ) ) ).
```

Listing 18: R postulate in FO^{TPC} using TPTP syntax

UR

UR postulate [BM06].

(UR) For $\omega_1 \in [\![\alpha]\!]$ and $\omega_2 \in [\![\neg\alpha]\!]$, either $\omega_1 <_{\Psi*\alpha} \omega_2$ or $\omega_2 <_{\Psi*\alpha} \omega_1$

```
fof('UR', postulate,
    ! [W1,W2] : (
        (
            int(W1)
            & int(W2)
            & mod(W1, A)
            & ~ mod(W2, A)
        )
        =>
        (
            strictless(W1, W2, op(E0, A))
            | strictless(W2, W1, op(E0, A))
        ) ) ).
```

Listing 19: UR postulate in FO^{TPC} using TPTP syntax

Darwiche and Pearl

Darwiche and Pearl postulates (CR1) - (CR6) [DP97] are shown in Listing 20.

```
fof('CR1', postulate,
    ! [W1,W2] : (
        (int(W1) & int(W2) & mod(W1, A) & mod(W2, A))
        =>
        (lesseq(W1, W2, E0) <=> lesseq(W1, W2, op(E0, A))) ) ).
fof('CR2', postulate,
    ! [W1,W2] : (
        (int(W1) & int(W2) & ~mod(W1, A) & ~mod(W2, A))
        =>
        (lesseq(W1, W2, E0) <=> lesseq(W1, W2, op(E0, A))) ) ).
fof('CR3', postulate,
    ! [W1,W2] : (
        (int(W1) & int(W2) & mod(W1, A) & ~mod(W2, A))
        =>
        (strictless(W1, W2, E0) => strictless(W1, W2, op(E0, A))) ) ).
fof('CR4', postulate,
    ! [W1,W2] : (
        (int(W1) & int(W2) & mod(W1, A) & ~mod(W2, A))
        =>
        (lesseq(W1, W2, E0) => lesseq(W1, W2, op(E0, A))) ) ).
fof('CR5', postulate,
    ! [W1,W2,W3] : (
        (int(W1) & int(W2) & int(W3) & mod(W1, A) & mod(W3, A) & ~ mod(
           W2, A))
        =>
        (
            (strictless(W3, W1, E0) & lesseq(W2, W1, E0))
            =>
            lesseq(W2, W1, op(E0, A))
        ) ) ).
fof('CR6', postulate,
    ! [W1,W2,W3] : (
        (int(W1) & int(W2) & int(W3) & mod(W1, A) & mod(W3, A) & ~ mod(
           W2, A))
        =>
        (
            (strictless(W3, W1, E0) & strictless(W2, W1, E0))
            =>
            strictless(W2, W1, op(E0, A))
        ) ) ).
```

Listing 20: Postulates from [DP97] in FO^{TPC} using TPTP syntax

Selbständigkeitserklärung

Ich erkläre, dass ich die Abschlussarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Unterschrift:

_____

Datum:

_____