PolyCoP: A Connection Prover For (Possibly) Any Logical Language*

Renan L. Fernandes[1], Fred Freitas[1], Ivan Varzinczak[2,3,4] and Pedro P.M. Farias[1,5]

[1]Centro de Informática, Universidade Federal de Pernambuco, Brazil
[2]LIASD, Université Paris 8, France
[3]CAIR, University of Cape Town, South Africa
[4]ISTI–CNR, Italy
[5]ARCE, Public Services Regulation Agency-CE, Brazil

Abstract

Automated Theorem Provers (ATPs) are generally complex and optimized systems with thousands or millions of code lines, making it difficult for beginners and researchers to understand and reuse ATPs for their own purposes. This paper presents PolyCoP, an ATP designed to implement a polymorphic version of the connection method that abstracts the need to code proof-related algorithms. PolyCoP allows the user to focus only on the (rather small) differences in literal representation, unification, and blocking strategies from each logic and to help logic beginners learn the connection method straightforwardly. Furthermore, the PolyCoP generates a proof tree as its proving algorithm that can be useful for understanding how the proof happened and representing it visually using a LaTeX system.

Keywords

Connection method, Automated Theorem Proving, Automated Reasoning

1. Introduction

Coding automated theorem provers (ATPs) can be arduous, depending on the logical language intended to be implemented. The main inferencing methods, such as resolution and tableaux, require complex implementations, typically involving thousands or millions of code lines.

Furthermore, they rely on specific optimization techniques that may work for a few logics or scenarios. Those implementations require people to have a strong background knowledge of logic, data structures, and algorithms. It can be a major handicap for beginners who want to learn about logic and ATP and researchers who want to explore ATP capabilities in their own work.

On the other hand, the Connection method (CM) [1] is a viable alternative as an ATP method. The CM is a goal-oriented proof search technique that checks whether a formula is valid, looking for the so-called connections, which are complementary pairs of literals.

Many connection calculi have been proposed for various logical languages, such as FOL [2, 3], modal [4, 5], intuitionistic [6, 7], paraconsistent [8] and description logics [9, 10, 11].

Besides, CM-proof systems are uniform, i.e., they cover different logical languages without significant change in their core implementation [12]. The calculi of those systems rely on the same structures and concepts, such as connection, copy, and blocking. However, the Connection Provers (CoP, the calculi implementation) do not take advantage of uniformity because they are implemented in different programming languages and are rooted in different algorithms — such as Otten's Prolog systems and

RuleML+RR'24: Companion Proceedings of the 8th International Joint Conference on Rules and Reasoning, September 16–22, 2024, Bucharest, Romania

*This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

rlf5@cin.ufpe.br (R. L. Fernandes); fred@cin.ufpe.br (F. Freitas); ivan.varzinczak@univ-paris8.fr (I. Varzinczak); ppmf@cin.ufpe.br (P. P.M. Farias)

0000-0001-9553-5515 (R. L. Fernandes); 0000-0003-0425-6786 (F. Freitas); 0000-0002-0025-9632 (I. Varzinczak); 0000-0002-6344-4448 (P. P.M. Farias)

Dimas et al. 's C++ implementation. If they share one core implementation, one high-level optimization implementation can benefit all provers simultaneously.

Thus, the objectives of this work are:

1. to develop a polymorphic connection method prover capable of proving numerous logics by reusing its core implementation; and
2. to implement a simple prover/framework allowing beginners to understand and use a connection method prover.

On the other hand, this work does not aim to provide an optimized and highly competitive ATP, as our goal is to show a simpler and general version of a CoP.

This paper presents PolyCoP, a Java connection method framework built using Java interfacing that isolates the code related to the proof search from the code related to the logical language such as:

- where two literals are complementary;
- how two complementary literals connect;
- how a clause is copied; and
- when the proof must be blocked.

The logical-related specifications are represented as strategies. Figure 1 gives a glimpse of the capability of PolyCoP. The connection method proof search has already been implemented. However, it requires code related to the target logical language to check whether a given formula is valid.

Thus, the user (someone who wants to develop an ATP for some logic or a beginner who wants to learn about provers) focuses only on a fragment of code related to the given logical language (for propositional logic or any other logic).

The input of the prover is a formula that needs to be mapped to a matrix by some user-specific code - a mapper. The matrix is the input of the prover, which must be linked to strategies to connect, copy or block the literals of the matrix according to the user specification. Those strategies establish how the prover uses the literals in the matrix.

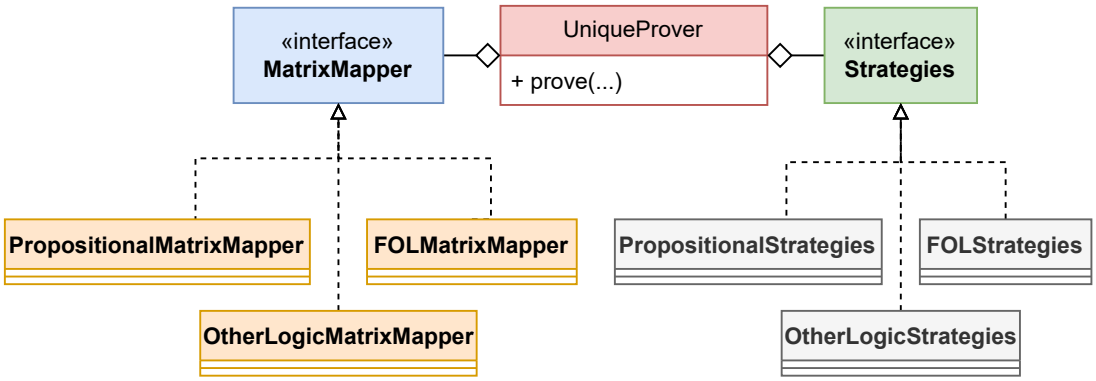

Figure 1: UML diagram of PolyCoP.

The following sections of the paper present a short background about CM (Section 2), our contribution: PolyCoP (Section 3), and concluding remarks (Section 4).

2. Background: the connection method

We start by presenting the connection method for propositional logic. The connection method is a direct proof method, i.e., it checks whether $\alpha \models \beta$ by checking whether $\neg\alpha \vee \beta$ is a tautology. The method requires propositions in Disjunctive Normal Form (DNF) as input; therefore, propositions

are disjunctions of conjunctions of *literals*, which are atomic propositions or their negation. The conjunctions of literals are called *clauses*, and the disjunction of clauses compounds the *matrix*.

Assume we want to check whether $A \wedge (\neg A \vee (\neg D \wedge (B \vee E) \wedge \neg E) \wedge (\neg B \vee \neg C) \models \neg C$. Its corresponding (positive[1]) DNF formula is $\neg A \vee (A \wedge D) \vee (A \wedge \neg B \wedge \neg E) \vee (A \wedge E)(B \wedge C) \vee \neg C$. A, $\neg A$, and so on are the literals, $\{A, \neg B\}$ is a clause, and, finally, $\{\{\neg A\}, \{A, D\}, \{A, \neg B, \neg E\}, \{A, E\}, \{B, C\}, \{\neg C\}\}$ is the matrix. Figure 2 illustrates the matrix in the graphical notation.

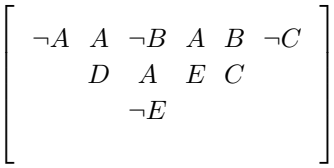

Figure 2: Matrix representation of $A \wedge (\neg A \vee (\neg D \wedge (B \vee E) \wedge \neg E) \wedge (\neg B \vee \neg C) \models \neg C$.

If we change our perspective and look at the matrix vertically, we see the *paths*, sets of literals containing only a literal of each clause. Figure 3 shows two of the 24 paths of the matrix. The left one is the path $\{\neg A, A, \neg B, A, B, \neg C\}$ and the right one is the path $\{\neg A, D, \neg B, A, C, \neg C\}$.

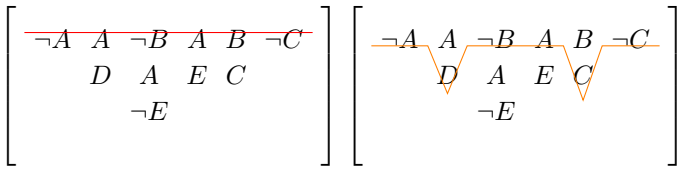

Figure 3: Graphical representation of two paths of the matrix. The left one is the path $\{\neg A, A, \neg B, A, B, \neg C\}$ and the right one is $\{\neg A, D, \neg B, A, C, \neg C\}$

The set of all paths of a matrix represents the same matrix formula in Conjunctive Normal Form (CNF). In other words, the two paths shown in Figure 3 are disjunctions that compound the formula in CNF.

So, the goal of the connection method is to check the paths, looking for complementary literals on them. A path containing a *complementary set* $\{L, \neg L\}$ is a disjunction containing $L \vee \neg L$, which is a tautology. If every matrix path contains a complementary set, the matrix is called *complementary* or *spanning*.

Our proof starts with the first clause, $\{\neg A\}$. We select its only literal and connect it with its complement in the second clause. After making this connection, we must check the remaining literals in both clauses. Since the first clause has only one literal, we have already checked it. Figure 4 shows the connection.

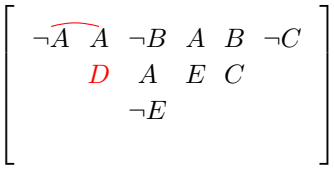

Figure 4: Proof started connecting $\neg A$ with the literal A in the second clause. However, backtracking is necessary once there is no complementary literal of D.

If we cannot find a complement for the literal we chose from the second clause (in this case, D), we backtrack our proof and undo the connection between the first and second clauses. Backtracking is necessary when the proof search gets stuck and cannot find a literal to connect.

We restart the proof by connecting the chosen literal from the first clause with its complement in the third clause (Figure 5a). The literals $\neg B$ and $\neg E$ still need to be checked. We then connect $\neg B$ with B

[1]The negation of the premise and preserving the consequence is called a positive representation of a formula.

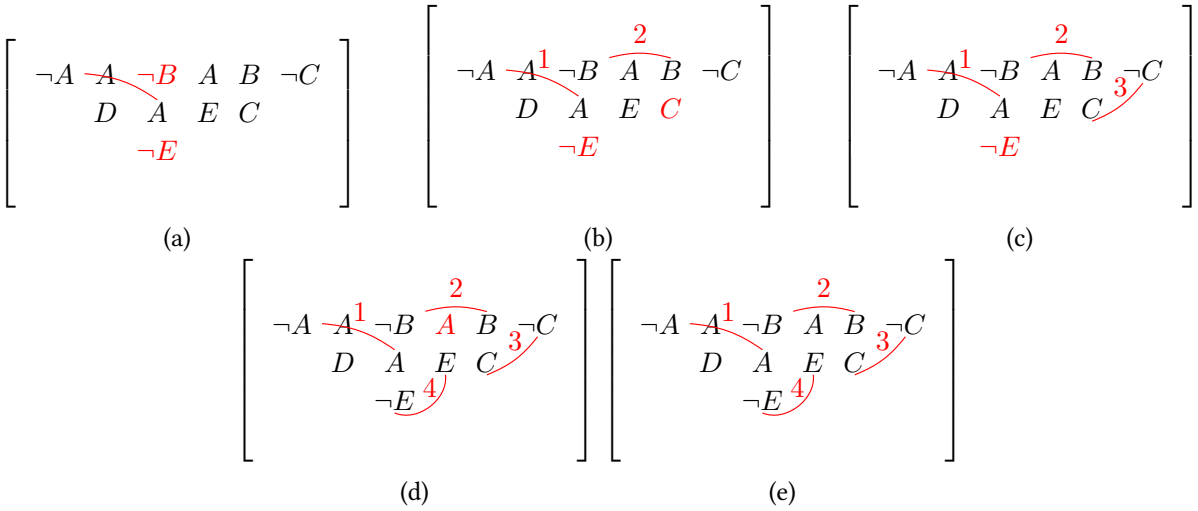

Figure 5: Connection method proof search. The remaining literals at each step are highlighted.

from the fifth clause (Figure 5b). Now, $\neg E$ and C remain to be checked.

After that, we connect C with $\neg C$ in the last clause (Figure 5c). Finally, we connect the last literal to be checked, $\neg E$, with E in the fourth clause (Figure 5d). At this point, the only remaining literal to be checked is A in the fourth clause. We can check it without connecting to a new clause because a complementary literal, $\neg A$, exists in the path (Figure 5e). We call this a "reduction".

Therefore, as all the paths have been verified to contain a connection, we conclude that $A \wedge (\neg A \vee (\neg D \wedge (B \vee E) \wedge \neg E) \wedge (\neg B \vee \neg C) \models \neg C$ is a tautology.

3. PolyCoP

PolyCoP, the general and polymorphic connection method written in Java, is available at GitHub[2]. The architecture relies on Java features, such as interfaces and generics, and the design patterns[13] strategy and state. The CoP implementation calls the strategies methods to prove the formula using their interfaces. So, the PolyCoP can run its proof search using different logic and algorithms by changing the interface implementations.

3.1. Strategies

There are four different strategy interfaces: `LiteralHelperStrategy`, `ConnectionStrategy`, `CopyStrategy`, and `BlockingStrategy`.

The strategy called `LiteralHelperStrategy` allows developers to define complementary literals and implement the logic behind them. This is essential for the prover to find complementary literals of a given one. The implementation takes a given literal L and the matrix M as input. The developer's code must return a submatrix of clauses M' where each clause has a complementary literal of L. The meaning of complementary can change based on the logic being implemented.

The `ConnectionStrategy` plays a crucial role in maintaining the correct behaviour of the code. It returns a Boolean value indicating whether the connection is valid. The inputs are the literals to be connected, the active path, and the current state of the strategy. The developer must check the current state and the path and update the strategy's current state to ensure the correct behaviour of the code. The state update must create another object representation of it instead of changing the properties of the current state. This allows for fast restoration of previous states when backtracking occurs.

As the name suggests, the `CopyStrategy` deals with copying clauses in the matrix. Some logic may require copying clauses to be reused in another subproof later. The input to this strategy is the clause

[2] https://github.com/renanlf/polycop

to be copied and the current state of the strategies. These inputs help the developer check whether a copy occurs.

Finally, the `BlockingStrategy` helps the developer determine when the proof needs to stop the search. It returns *true* when the proof needs to stop the search after a connection and *false* otherwise. The input to this strategy is the current state of the other strategies. The developer can use this strategy to check the soundness of the proof after a connection instead of checking it before the connection happens.

In Figure 6, we show the detailed UML class diagram of PolyCoP.

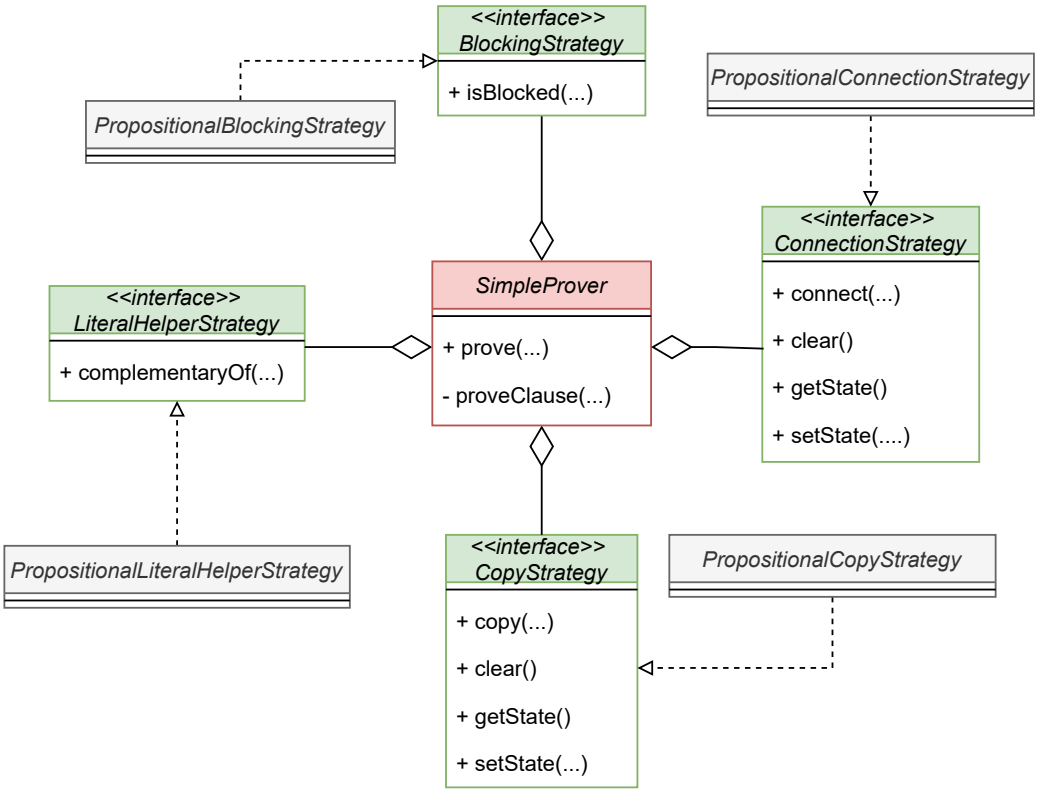

Figure 6: Detailed UML class diagram of PolyCoP strategies and the prover

3.2. Implementation for propositional logic

To illustrate the PolyCoP polymorphism, we show the implementation of the well-known propositional logic in this paper. The main reason for using it as PolyCoP's starter logic is to focus the reader on the system features without requiring a background in other logic.

Nevertheless, this decision does not restrict PolyCoP to simple logic. The code repository also contains other implementations for first-order and defeasible Description Logic.

Our implementation's input is files in the well-known CNF format. It represents the literals as integers, where a negative number $-N$ denotes the negation of the literal N. Because it is already a clausal representation, we straightforwardly translate the values to the matrix, multiplying them by -1, to obtain the DNF version. The intuition is that once we negate the formula in CNF (because CM is a direct method), it is the same as changing the polarity of its literals.

Delving deeper into the data structures of PolyCoP, we use linked nodes to store the states through the search proof. The main reason to do so is that this approach optimizes memory usage when the proof tree becomes deeper.

To illustrate the memory benefits, assume the code stores a new object every time a proof rule is applied. So, proving some literal in a 10-depth proof tree in an immutable stateful scenario must contain states with 0 to 9 objects each. The code must store 37 references to represent those ten states.

However, the number of references needed decreases if the code uses linked nodes to store the states. The 10th state only needs to store one object's reference and the previous linked node's reference. The previous linked node also requires two references, and so on. The proof must store 18 references to represent the same ten states.

Furthermore, the connection proof result is a tree where each node represents a different tuple of the connection proof search. The root is the Start Rule, and each rule's application appends new children to the nodes. Such an approach is helpful for human-readable search proofs.

Implementing strategies for a logic turns PolyCoP into a prover of that logic. Thus, implementing strategies for first-order logic (fol), for example, makes PolyCoP a first-order logic prover. Such an implementation must have a connection strategy that checks the substitution states in the path to validate a unification. It also must ensure the blocking strategy copies terms properly.

The blocking strategy, however, can be ignored due to the nature of the undecidable first-order logic formulae. One can also implement a blocking strategy that stops when it reaches some condition to prune the proof search. Those different strategies are possible in PolyCoP, and users can try whatever approach they want.

Thus, for PolyCoP to be a $\mathcal{ALC}$[14] prover, for example, the blocking strategy must map to $true$ at some point once $\mathcal{ALC}$ is a decidable fragment of description logics. Looking to the $\mathcal{ALC}\theta$ - CM [9], it must happen when the blocking condition [9] holds it.

3.3. LaTeX outputs

Lastly, besides the core implementation of PolyCoP, a helpful feature is that it outputs the proof as LaTeX code. Our code provides an implementation that translates the connection proof tree as a LaTeX string, allowing the user to represent the connection proof visually in a sequential style.

Following the example of Section 2, Figure 7 shows the connection proof interpreted by LaTeX systems.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\}, M, [1,2,-3]}\,\text{Ax} \quad \dfrac{}{\{\}, M, [1,2]}\,\text{Ax}}{[-3], M, [1,2]}\,Ext \quad \dfrac{}{\{\}, M, [1]}\,\text{Ax}}{[2], M, [1]}\,Ext \quad \dfrac{}{\{\}, M, []}\,\text{Ax}}{[1], M, []}\,Ext}{\varepsilon, M, \varepsilon}\,St$$

Figure 7: Proof in sequent-style interpreted by a LaTeX system of PolyCoP generated proof.

Listing 1 shows a fragment of the code generated by PolyCoP. Using PolyCoP, the user can export it to a file and generate the corresponding sequent-style proof.

```
\documentclass[...]{standalone}
\usepackage{bussproofs} % for sequent-style proofs
\begin{document}
\AxiomC{}
\RightLabel{Ax}
\UnaryInfC{$\{\}, M, [1]$}
...
\AxiomC{}
\RightLabel{Ax}
\UnaryInfC{$\{\}, M, []$}
```

```
\RightLabel{\textit{Ext}}
\BinaryInfC{$[-2], M, []$}
\RightLabel{\textit{Ext}}
\BinaryInfC{$[1, -2], M, []$}
\RightLabel{\textit{St}}
\UnaryInfC{$\varepsilon, M, \varepsilon$}

\DisplayProof

\end{document}
```

Listing 1: Fragment of code generated by PolyCoP of Example's proof.

4. Conclusion

In this paper, we presented the PolyCoP, a general CM prover that benefits from the uniformity of CMs implementing a single prover to possibly any logical language.

This is the first implementation of the general connection prover. However, further optimized implementations will easily reuse the Java strategy interfaces without any change. Take the lemmata optimization [15], for instance. It reuses subproofs of literals if they already have been proved in the proof search. We can provide such behaviour by adding a new structure to store the proofs and look for them each time PolyCoP calls the proof method. If N logical language strategies were implemented, the optimization would improve the proof search for the N logical languages.

Another easy win is to extend the implementation to a multi-thread approach once immutable states can be shared with less or no effort between the threads during the proof search. On the other hand, low-level optimization and logical language full implementations such as leanCoP [2] and Raccoon [16] should be more efficient.

As a first version, we understand that a few changes may be necessary until a stable general connection prover version is obtained. In future work, we can explore the best design decisions, such as the best data structures, from a benchmark and thus check whether a better approach for implementing the general prover.

The next step of this work is to provide more provers and logical language strategies to illustrate the general approach and create benchmarks for different cases, such as CASC and ORE competitions, using the very same prover.

References

[1] W. Bibel, Automated Theorem Proving, Vieweg Verlag, Wiesbaden, 1987.

[2] J. Otten, W. Bibel, leancop: lean connection-based theorem proving, J. Symb. Comput. 36 (2003) 139–161.

[3] J. Otten, A non-clausal connection calculus, in: K. Brünnler, G. Metcalfe (Eds.), Automated Reasoning with Analytic Tableaux and Related Methods - 20th International Conference, TABLEAUX 2011, Bern, Switzerland, July 4-8, 2011. Proceedings, volume 6793 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 226–241.

[4] J. Otten, Implementing connection calculi for first-order modal logics, in: K. Korovin, S. Schulz, E. Ternovska (Eds.), IWIL 2012: The 9th International Workshop on the Implementation of Logics, Merida, Venezuela, March 10, 2012, volume 22 of *EPiC Series in Computing*, EasyChair, 2012, pp. 18–32.

[5] J. Otten, Advancing automated theorem proving for the modal logics D and S5, in: C. Benzmüller, J. Otten (Eds.), Proceedings of the 4th International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2022) affiliated with the 11th International Joint Conference on

Automated Reasoning (IJCAR 2022), Haifa, Israel, August 11, 2022, volume 3326 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 81–91.

[6] J. Otten, Non-clausal connection-based theorem proving in intuitionistic first-order logic, in: C. Benzmüller, J. Otten (Eds.), Proceedings of the 2nd International Workshop Automated Reasoning in Quantified Non-Classical Logics (ARQNL 2016) affiliated with the International Joint Conference on Automated Reasoning (IJCAR 2016)., Coimbra, Portugal, July 1, 2016, volume 1770 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016, pp. 9–20.

[7] J. Otten, Non-clausal connection calculi for non-classical logics, in: R. A. Schmidt, C. Nalon (Eds.), Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25-28, 2017, Proceedings, volume 10501 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 209–227.

[8] D. Krause, E. F. Nobre, M. A. Musicante, Bibel's matrix connection method in paraconsistent logic: General concepts and implementation, in: 21st International Conference of the Chilean Computer Science Society (SCCC 2001), 6-8 November 2001, Punta Arenas, Chile, IEEE Computer Society, 2001, pp. 161–167.

[9] F. Freitas, J. Otten, A Connection Calculus for the Description Logic $\mathcal{ALC}$, in: 29th Canadian Conference on Artificial Intelligence, Springer, Victoria, BC, Canadá, 2016, pp. 243–256.

[10] F. Freitas, I. Varzinczak, Cardinality restrictions within description logic connection calculi, in: C. Benzmüller, F. Ricca, X. Parent, D. Roman (Eds.), Rules and Reasoning - Second International Joint Conference, RuleML+RR 2018, Luxembourg, September 18-21, 2018, Proceedings, volume 11092 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 65–80.

[11] R. Fernandes, F. Freitas, I. Varzinczak, A connection method for a defeasible extension of ALC, in: M. Homola, V. Ryzhikov, R. A. Schmidt (Eds.), Proceedings of the 34th International Workshop on Description Logics (DL 2021) part of Bratislava Knowledge September (BAKS 2021), Bratislava, Slovakia, September 19th to 22nd, 2021, volume 2954 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.

[12] W. Bibel, A vision for automated deduction rooted in the connection method, in: R. A. Schmidt, C. Nalon (Eds.), Automated Reasoning with Analytic Tableaux and Related Methods - 26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25-28, 2017, Proceedings, volume 10501 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 3–21.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Pearson Education, 1994.

[14] S. Cai, Z. Ming, S. Li, BALC: A belief extension of description logic ALC, in: 9th International Conference for Young Computer Scientists, IEEE, Hunan, China, 2008, pp. 1711–1716.

[15] J. Otten, Restricting backtracking in connection calculi, AI Commun. 23 (2010) 159–182.

[16] D. Melo Filho, F. Freitas, J. Otten, RACCOON: A Connection Reasoner for the Description Logic $\mathcal{ALC}$, in: 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, volume 46, EasyChair, Maun, Botswana, 2017, pp. 200–211.