An API for DL Abduction Solvers

Zuzana Hlávková[1], Martin Homola[1], Patrick Koopmann[2] and Júlia Pukancová[1]

[1]Comenius University in Bratislava, Mlynská dolina, 842 41 Bratislava, Slovakia
[2]Theoretical Computer Science, TU Dresden, Dresden, Germany

Abstract
As abduction is getting more attention in the world of ontologies, multiple abduction solvers for description logics (DL) have been developed. So far, however, there was no attempt for a unified API that would facilitate the integration of different DL abduction solvers in an application, in the way e.g. the OWL API does it for deductive OWL reasoning systems. In order to fill this gap, we abstract the common functionalities of different DL abduction solvers and introduce the DL Abduction API.

Keywords
abduction, description logics, ontologies, software engineering

1. Introduction

Abduction, stemming from the ideas of Peirce [1], is a reasoning task to provide hypothetical explanations of why some observations of a modelled phenomenon are *not supported* by deductive entailments of a knowledge model of the phenomenon. Specifically in DL, given a knowledge base $\mathcal{K}$ and an *observation* in form of a set of axioms $\mathcal{O}$ s.t. $\mathcal{K} \not\models \mathcal{O}$, we are looking for *explanations* in form of sets of axioms $\mathcal{E}$ that one can add to $\mathcal{K}$ to support $\mathcal{O}$, that is, for which $\mathcal{K} \cup \mathcal{E} \models \mathcal{O}$ [2]. For example, consider a knowledge base $\mathcal{K}$:

$$\mathsf{Mother} \sqcup \mathsf{Father} \sqsubseteq \mathsf{Parent}$$
$$\mathsf{Parent} \sqsubseteq \mathsf{Happy}$$
$$\mathsf{Person(Jack)}, \quad \mathsf{Parent(Jill)}$$

We may explain the observation $\mathcal{O}_1 = \{\mathsf{Happy(jack)}\}$ by any of the following explanations: $\mathcal{E}_1 = \{\mathsf{Mother(jack)}\}$, $\mathcal{E}_2 = \{\mathsf{Father(jack)}\}$, $\mathcal{E}_3 = \{\mathsf{Parent(jack)}\}$.

Depending on the type of axioms which are allowed in $\mathcal{O}$, and for which we are looking in $\mathcal{E}$, we distinguish *ABox abduction* (as in our example above) which looks for explanations on the data level, i.e. providing explanations as ABox axioms [3, 4, 5, 6].

In turn, *TBox abduction* looks for explanations on the conceptual level, i.e. providing explanations as TBox axioms [7, 8, 9]. Assume again $\mathcal{K}$ as above, then $\mathcal{O}_2 = \{\mathsf{Mother} \sqsubseteq \mathsf{Person}\}$ may be explained by $\mathcal{E}_4 = \{\mathsf{Parent} \sqsubseteq \mathsf{Person}\}$.

DL 2022: 35th International Workshop on Description Logics, August 7–10, 2022, Haifa, Israel
hlavkovazuz@gmail.com (Z. Hlávková); homola@fmph.uniba.sk (M. Homola); patrick.koopmann@tu-dresden.de (P. Koopmann); pukancova@fmph.uniba.sk (J. Pukancová)
https://dai.fmph.uniba.sk/~homola/ (M. Homola); https://lat.inf.tu-dresden.de/~koopmann/ (P. Koopmann); https://dai.fmph.uniba.sk/~pukancova/ (J. Pukancová)
0000-0001-6384-9771 (M. Homola); 0000-0001-5999-2583 (P. Koopmann)

More generally, *knowledge base abduction* is not constrained to ABox or TBox axioms [10, 2]. For instance, $\mathcal{O}_1$ may also be explained w.r.t. $\mathcal{K}$ by $\mathcal{E}_5 = \{\mathsf{loves}(\mathsf{jack},\mathsf{jill}), \mathsf{Parent} \sqsubseteq \mathsf{Person}, \exists\mathsf{loves}.\mathsf{Person} \sqsubseteq \mathsf{Happy}\}$.

The definition of abduction outlined above provides the basic semantic framework for establishing what is an explanation of a given problem. On the other hand, if one does not further constrain possible explanations, there may be too many. In fact, due to the monotonicity of standard DL, if there is one explanation, then there are already infinitely many in the general sense. Explanations are therefore often constrained to be *minimal.* This may be either considered in a syntactic sense (subset minimal), e.g. all $\mathcal{E}_{1-3}$ are subset minimal explanations of $\mathcal{O}_1$ while on the other hand $\mathcal{E}_6 = \{\{\mathsf{Parent}(\mathsf{jack}), \{\mathsf{Person}(\mathsf{jack})\}$ is not, as $\mathcal{E}_3 \subsetneq \mathcal{E}_6$ is smaller.

Or in a more refined semantic sense where only semantically weakest[1] explanations are considered [2], e.g. out of $\mathcal{E}_{1-3}$, only $\mathcal{E}_3$ is semantically minimal, as $\mathcal{K} \cup \mathcal{E}_1 \models \mathcal{E}_3$ and $\mathcal{K} \cup \mathcal{E}_2 \models \mathcal{E}_3$, but not the other way around.

Other relevant constraints which are almost always assumed are *consistency* ($\mathcal{K} \cup \mathcal{E}$ is consistent), *relevance* ($\mathcal{E} \not\models \mathcal{O}$), and *explanatoriness* ($\mathcal{K} \not\models \mathcal{O}$). However, depending on the application, different additional constraints may be useful, e.g. *solipsisticity* ($\mathcal{E}$ only contains individuals from $\mathcal{O}$). All explanations illustrated above are consistent, relevant and explanatory, and all but $\mathcal{E}_5$ are also solipsistic. For more details refer to Elsenbroich et al. [2].

Another way how to constrain the explanations – and thus the search space for the abduction reasoner – is by constraining the set of expressions that could possibly become explanations. This is done by specifying the *abducibles.* Here, abducibles may either refer to a restricted set of axioms of which the explanation has to be a subset [12, 13], or to a restricted signature of individual, concept and role names [14, 4], in which case there is still an unbounded set of axioms that can be used in an explanation. Indeed, given a particular application, the user may only be interested in explanations involving a certain specific set of concepts, roles or individuals. If such constraints are known beforehand, it may significantly improve the reasoner's running time, but they can also impact the computational complexity of abduction negatively [4].

Abduction in DL has a number of interesting applications, e.g. ontology debugging and support for test-driven ontology development [15], manufacturing control [16], medical diagnosis [17], multimedia interpretation [11], ontology repair [9] and explaining missing entailments [18, 14, 13].

A number of DL abduction solvers have been developed, including the works of Du et al. [12], Del-Pinto and Schmidt [19, 20], Pukancová and Homola [21], and Homola et al. [22], and Koopmann et al. [10]. Each of the solvers provides its own interface to the user (most often a command-line interface).

To our best knowledge, there was no attempt so far to specify a unified API interface, that could be used to integrate one of the solvers into an application that needs to use abductive reasoning – much in the fashion of how the popular OWL API [23] covers this task for deductive reasoning.

We propose the DL Abduction API with the aim to fill this gap. Similarly to the OWL API, it is implemented in Java. Once an abduction solver implements the API, any Java application can easily integrate it to compute answers for abduction problems over ontologies. The API

[1]Depending on the application, in some cases semantically strongest explanations may be preferred [11].

encapsulates the most common abduction inputs such as the input ontology, observations and abducibles, and includes switches for other common options. As most of the inputs and outputs are in fact OWL ontologies, axioms, and symbols, the OWL API is used for their handling, conveniently for developers already acquainted with it.

2. Abduction Solvers

A DL abduction solver needs to process inputs and provide outputs. We have analyzed several stand-alone solvers (mainly AAA [21], MHS-MXP [22, 24], and LETHE [10, 4]), and herein we summarize their main common characteristics:

Input ontology: Also called background knowledge, this is usually specified as a set of DL axioms in form of an OWL ontology.

Observations: Some solvers support observations that are single axioms [20, 7], while others support observations that consist of several axioms [21, 10].

Abducibles: The simplest approach is to specify abducibles by giving a set of abducible axioms. Hypotheses are then generated by picking appropriate subsets of the given set of abducible axioms [24]. In contrast, in signature-based abduction, abducibles are provided in form of a set of concept and role names, so that hypotheses are required to only use those abducible names provided, but can combine those names to build respective axioms [21, 24], including possibly complex ones [10] in arbitrary ways. Finally, approaches like [21, 24] allow to give further constraints on the shape of the axioms in addition to the signature restriction, for example by allowing loops in role assertions, allowing only concept names rather than complex concepts, or only allowing concepts of the form A and $\neg A$ for an atomic concept A.

Outputs: Each abduction solver computes one or several solutions, called hypotheses or explanations, which usually consist of a single axiom or a set of axioms. For some approaches, the space of solutions is potentially infinite, or can at least get very large [4], and the computation of solutions can potentially take a long time.

In addition to these, even if the proposed API is very broad in trying to incorporate all possible features, there might always be some additional **internal settings** and **debug outputs** and it might be desired to allow these to be handled by the API in a solver specific format. Thus, an API for DL abduction needs to address the following challenges:

- find a uniform way of representing abducibles, while supporting the different types of abducibles each solver may support;
- not only specify abducibles by means of axiom sets or signatures, but also allow to give specific restrictions on the syntactic shape;
- be flexible regarding the shape of axioms allowed in observations and explanations: in TBox abduction, those are restricted to be TBox axioms; in ABox abduction, they have to be ABox axioms, and finally, in KB abduction, there is no restriction at all;
- deal with potentially long computation times;
- deal with potentially large to infinite solution sets.

3. DL Abduction API

We implemented the proposed DL Abduction API in Java using OWL API [23]. A UML class diagram of the central classes and interfaces is shown in Figure 1. In order to support the API with their abduction library, developers have to implement the interfaces `AbductionManager`, `AbducibleContainer` and `AbductionManagerAndAbducibleContainerFactory`. The `AbductionManager` handles the main abduction process: here one specifies background knowledge, the observation and abducibles, and starts the abduction process. Since the computation of hypotheses can take time, it is possible to use the abduction manager in an asynchronous manner. For this, the user registers an abduction `Monitor`, which receives the hypotheses from the abduction manager once they are computed. For the abducibles, we offer a range of settings restricting the shape of axioms that can be used in the explanation. This is managed by `AbducibleContainer`, which stores information about abducible axioms, abducible signatures, and additional properties on the shape of axioms. To deal with the different types of axioms that are allowed in the abducibles, explanations and hypotheses, we use generic types, which allow to parametrize the type for instance to only allow for TBox axioms as explanations and observations.

In the following, we illustrate the usage of the API step-by-step, where for simplicity, we do not showcase the use of generics.

3.1. Basic Initialization

To instantiate `AbductionManager` and `AbducibleContainer`, the user uses the interface `AbductionManagerAndAbducibleContainerFactory` as implemented by the respective abduction library. The initialization is as follows:

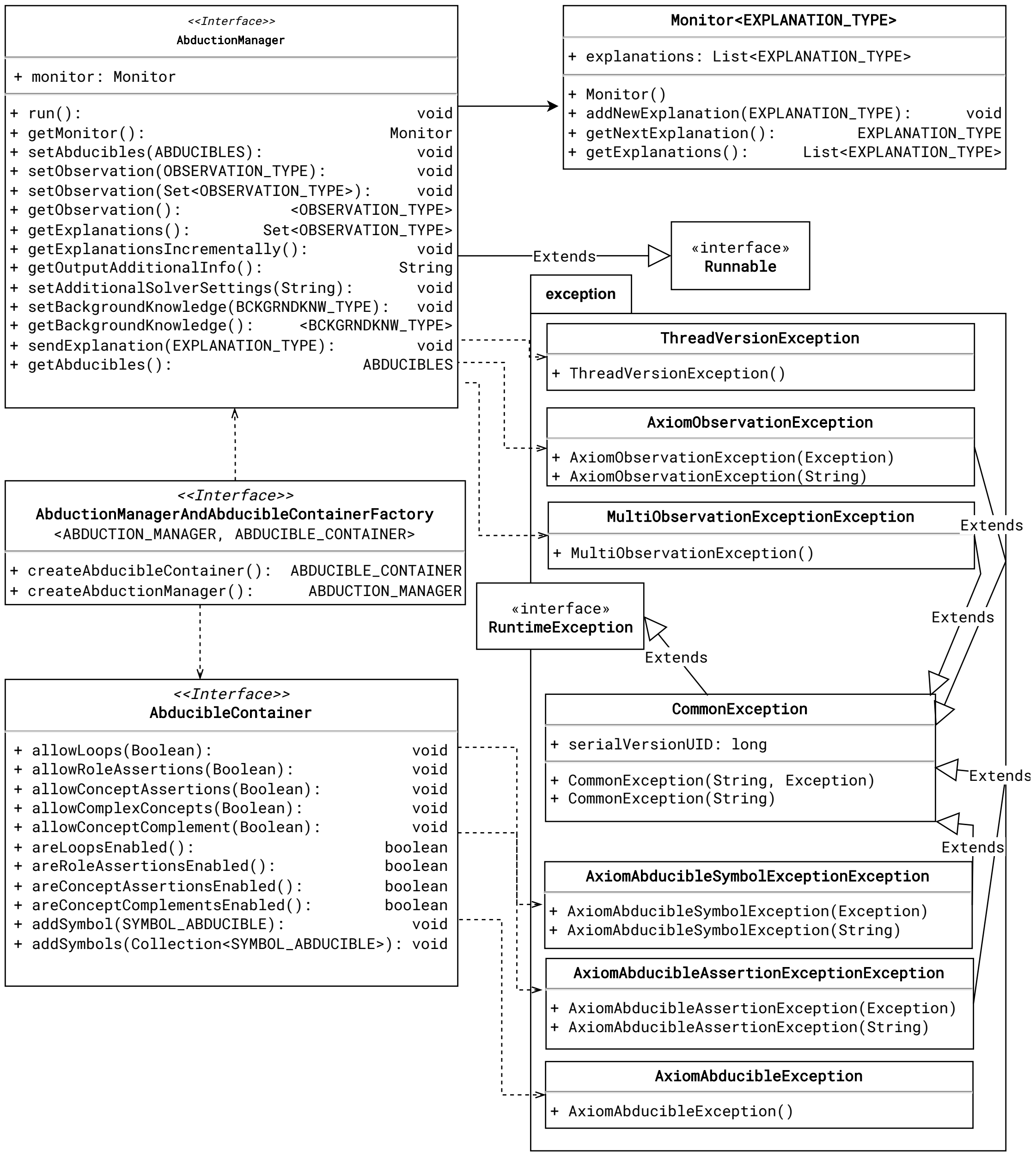
```
AbductionManagerAndAbducibleContainerFactory abductionFactory
  = new AbductionManagerAndAbducibleContainerFactoryImpl();
AbductionManager abductionManager
  = abductionFactory.createAbductionManager();
AbducibleContainer abducibleContainer
  = abductionFactory.createAbducibleContainer();
abductionManager.setAbducibles(abducibleContainer);
```

We then use the `AbductionManager` instance to configure the background ontology w.r.t. which we will perform the abduction task. The ontology is initialized and loaded via OWL API via an `OWLOntologyManager` instance and it is loaded from an IRI.

```
OWLOntologyManager man = OWLManager.createOWLOntologyManager();
IRI bgOIRI = IRI.create("http://example.org/ontology");
OWLOntology bgOntology = man.loadOntology(bgOIRI);

abductionManager.setBackgroundKnowledge(bgOntology);
```

We also use the `AbductionManager` instance to specify the observation. Most abduction reasoners accept observations in the form of a single axiom (which we will treat a singleton set) or set of axioms. Observations consisting of several axioms may not be supported and also not all possible forms of axioms may be accepted by the given reasoner. If the user supplies an



<<Interface>>
AbductionManager
+ monitor: Monitor
+ run(): void
+ getMonitor(): Monitor
+ setAbducibles(ABDUCIBLES): void
+ setObservation(OBSERVATION_TYPE): void
+ setObservation(Set<OBSERVATION_TYPE>): void
+ getObservation(): <OBSERVATION_TYPE>
+ getExplanations(): Set<OBSERVATION_TYPE>
+ getExplanationsIncrementally(): void
+ getOutputAdditionalInfo(): String
+ setAdditionalSolverSettings(String): void
+ setBackgroundKnowledge(BCKGRNDKNW_TYPE): void
+ getBackgroundKnowledge(): <BCKGRNDKNW_TYPE>
+ sendExplanation(EXPLANATION_TYPE): void
+ getAbducibles(): ABDUCIBLES
Monitor<EXPLANATION_TYPE>
+ explanations: List<EXPLANATION_TYPE>
+ Monitor()
+ addNewExplanation(EXPLANATION_TYPE): void
+ getNextExplanation(): EXPLANATION_TYPE
+ getExplanations(): List<EXPLANATION_TYPE>
Extends
«interface»
Runnable
exception
ThreadVersionException
+ ThreadVersionException()
AxiomObservationException
+ AxiomObservationException(Exception)
+ AxiomObservationException(String)
MultiObservationExceptionException
+ MultiObservationException()
Extends
Extends
<<Interface>>
AbductionManagerAndAbducibleContainerFactory
<ABDUCTION_MANAGER, ABDUCIBLE_CONTAINER>
+ createAbducibleContainer(): ABDUCIBLE_CONTAINER
+ createAbductionManager(): ABDUCTION_MANAGER
«interface»
RuntimeException
Extends
CommonException
+ serialVersionUID: long
+ CommonException(String, Exception)
+ CommonException(String)
Extends
Extends
<<Interface>>
AbducibleContainer
+ allowLoops(Boolean): void
+ allowRoleAssertions(Boolean): void
+ allowConceptAssertions(Boolean): void
+ allowComplexConcepts(Boolean): void
+ allowConceptComplement(Boolean): void
+ areLoopsEnabled(): boolean
+ areRoleAssertionsEnabled(): boolean
+ areConceptAssertionsEnabled(): boolean
+ areConceptComplementsEnabled(): boolean
+ addSymbol(SYMBOL_ABDUCIBLE): void
+ addSymbols(Collection<SYMBOL_ABDUCIBLE>): void
AxiomAbducibleSymbolExceptionException
+ AxiomAbducibleSymbolException(Exception)
+ AxiomAbducibleSymbolException(String)
AxiomAbducibleAssertionExceptionException
+ AxiomAbducibleAssertionException(Exception)
+ AxiomAbducibleAssertionException(String)
AxiomAbducibleException
+ AxiomAbducibleException()

Figure 1: Class diagram of DL Abduction API highlighting central methods; type parameters for AbductionManager and AbducibleContainer omitted

observation that is not supported, the abduction manager will throw `MultiObservationException` or `AxiomObservationException` respectively.

```
IRI obsOIRI = IRI.create("http://example.org/observations");
OWLOntology obsOntology = man.loadOntology(obsOIRI);
Set<OWLOntology> obsOntologySet = new HashSet<>();
```

```
obsOntologySet..add(obsOntology);
try {
  abductionManager.setObservation(obsOntologySet);
} catch (CommonException ex) {
  throw new CommonException("Solver exception: ", ex);
}
```

3.2. Configuring Abducibles

Abducibles are vital to constrain the solution space. They can be specified in multiple ways. The first option is to specify possible symbols from which possible explanations may be constructed (in which case we are essentially performing signature-based abduction). For this we initialize an instance of and `OWLDataFactory`. Consecutively the `abducibleContainer` is configured by adding all these OWL API entity representations.

```
OWLDataFactory df = o.getOWLOntologyManager().getOWLDataFactory();

OWLIndividual indJack = df.getOWLNamedIndividual(bgOIRI+"#jack");
OWLIndividual indJill = df.getOWLNamedIndividual(bgOIRI+"#jill");
OWLClass clsParent = df.getOWLClass(bgOIRI+"#Parent");
OWLObjectProperty oprHasChild = df.getOWLObjectProperty(bgOIRI+"#
    hasChild");

try {
    abducibleContainer.addSymbol(indJack);
    abducibleContainer.addSymbol(indJill);
    abducibleContainer.addSymbol(clsParent);
    abducibleContainer.addSymbol(oprHasChild);
} catch (CommonException ex) {
    throw new CommonException("Solver exception: ", ex);
}
```

Alternatively, we allow passing of abducible symbols via an ontology that contains declarations of all abducible symbols.

```
OWLOntology abdSymbolList =
  man.loadOntologyFromOntologyDocument(new File("abd-symbols.owl"));

try {
    abducibleContainer.addSymbols(abdSymbolList);
} catch (CommonException ex) {
    throw new CommonException("Solver exception: ", ex);
}
```

The `addSymbol` and `addSymbols` methods may throw an `AxiomAbducibleSymbolException` in case the passed symbols are not supported by the respective abduction solver.

In addition to specifying the signature, `AbducibleContainerImpl` features a number of Boolean switches to control the shape of the axioms in a solution. `allowConceptAssertions` and `allowRoleAssertions` respectively allow or disallow axioms in the hypotheses that are concept or role assertions. If concept assertions are allowed, `allowConceptComplement` and `allowComplexConcepts` can be used to determine whether only concept names, negated concept names, or also complex concepts are allowed. With role assertions enabled, `allowLoops` may be used to toggle reflexive role assertions (loops).

Alternatively to the option above (i.e. to specify abducible symbols, possibly supplemented by constraints on the generated axioms), users may also directly specify the set of abducible axioms. Both approaches are mutually exclusive: if one tries to specify abducible symbols and abducible axioms, the API will throw an exception. Similarly it is not possible to specify abducible axioms and additional constraints on their shape (as constraints only apply to axioms generated from abducible symbols).

An example of the latter option (i.e. abducible axioms) follows:

```
OWLIndividual indJack = df.getOWLNamedIndividual(bgOIRI+"#jack");
OWLClass clsPerson = df.getOWLClass(bgOIRI+"#Person");
OWLClass clsParent = df.getOWLClass(bgOIRI+"#Parent");
OWLObjectComplementOf clsC1 = df.getOWLObjectComplementOf(clsParent);
OWLObjectIntersectionOf clsC2
  = df.getOWLObjectIntersectionOf(clsPerson, clsC1);

try {
  abducibleContainer.addAssertion(
    df.getOWLClassAssertionAxiom(clsPerson, indJack));
  abducibleContainer.addAssertion(
    df.getOWLClassAssertionAxiom(clsParent, indJack));
  abducibleContainer.addAssertion(
    df.getOWLClassAssertionAxiom(clsC1, indJack));
  abducibleContainer.addAssertion(
    df.getOWLClassAssertionAxiom(clsC2, indJack));

} catch (CommonException ex) {
  throw new CommonException("Solver exception: ", ex);
}
```

Similarly as for the abducible signatures, it is possible to add several assertions at once by providing an OWLOntology object. Then, all axioms of that ontology are added.

```
OWLOntology abdAxiomList =
  man.loadOntologyFromOntologyDocument(new File("abd-axioms.owl"));

try {
  abducibleContainer.addAssertions(abdAxiomList);
} catch (CommonException ex) {
  throw new CommonException("Solver exception: ", ex);
}
```

Depending on the solver implementation, if e.g. an unsupported abducible axiom is passed then `AxiomAbducibleAssertionException` is thrown.

3.3. Internal Solver Settings

Different solvers may have additional specific functionalities which are not covered by our API. While this being the case it may still be useful to pass control parameters into the solver. We include a method to pass such information as a single string in a format prescribed by the given solver:

```
abductionManager.setAdditionalSolverSettings("internalSettings");
```

3.4. Running the Solver

Once everything is configured, users can run the solver and compute explanations, for which we again support different methods. If the number of solutions is finite, the following method can be used to compute all explanations:

```
Set<Explanation> explanations = abductionManager.getExplanations();
```

If one only requires single explanation, i.e. one that is found first, this can be computed as follows:

```
Explanation explanation = abductionManager.getExplanation();
```

The solvers may output additional information associated with the explanations (debug logs, etc.). This is also accessible via the API as follows:

```
String log = abductionManager.getOutputAdditionalInfo();
```

The search for all abductive explanations is computationally hard, and the implementation of the solver may in fact find some explanations early on, while it may take much longer to completely search through the whole search space. In order to make the explanations accessible on-the-fly, as soon as they are computed DL Abduction API implements a multi-threaded version based on the monitor design pattern. The respective UML sequential diagram in printed in Fig. 2.

First the monitor that handles the communication is obtained from the `abductionManager` instance:

```
monitor = abductionManager.getMonitor();
```

Then, in order to process the explanations as they are obtained, we initialize a second thread in which we iteratively query the monitor for new explanations:

```
Thread ct = new Thread() {
  public void run() {
    while (true) {
      synchronized(monitor) {
        try {
          monitor.wait();
          Object explanation = monitor.getNextExplanation();
          if (explanation == null) {
            monitor.notify();
            break;
          }
          // process the explanation as needed

          monitor.notify();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }
};

ct.start();
```

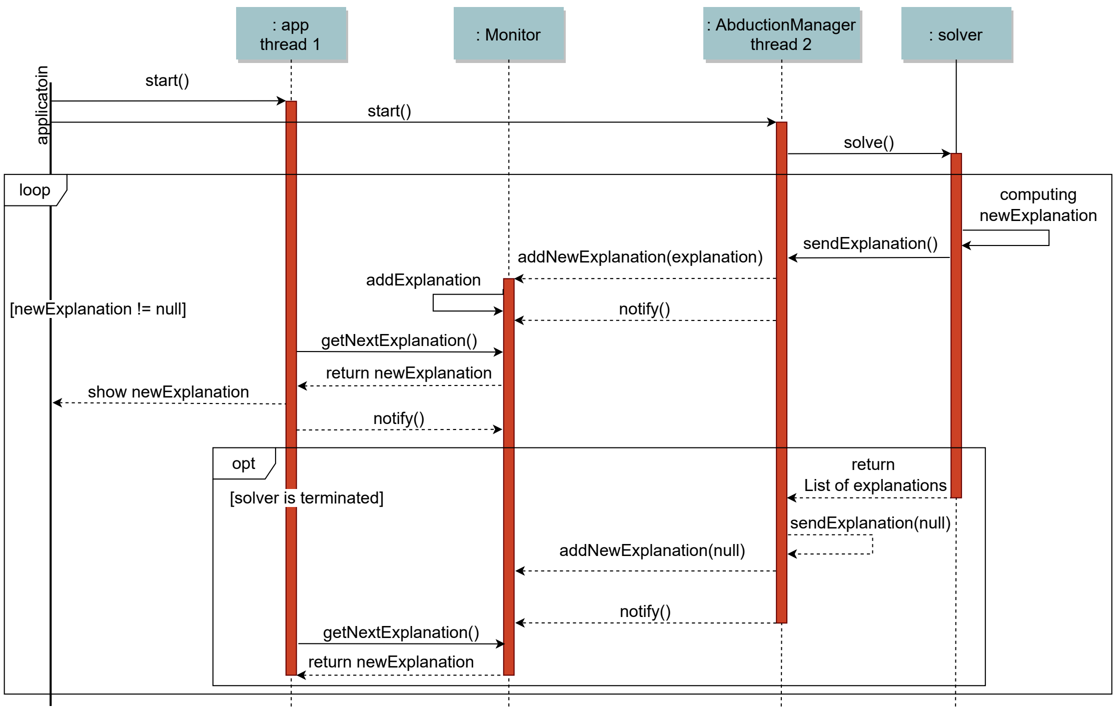


Figure 2: UML sequential diagram for on-the-fly explanation access

Then abduction is called by the method `getExplanationsIncrementally` and a new the computation is started in a new thread in the abductionManager:

```
abductionManager.getExplanationsIncrementally();
```

Once the solver's search for new explanations is over, the `monitor.getNextExplanation()` call returns `null` and the processing loop will break.

4. Implementation in MHS-MXP Solver

The DL Abduction API has been implemented into MHS-MXP [22, 24] which is an ABox abduction reasoner. The API functionality integrated includes all main abduction inputs: passing the background ontology, the observation (including a set of axioms), and abducibles.

Signature-based abducibles are supported by passing symbols (individuals and concepts), and the switch to allow or disallow negated concept assertions in explanations is implemented too. Axiom-based abducibles are also supported. (These two options are exclusive.) A depth-limitation for the MHS-tree and a time out can be passed as internal settings.

The solver may be run via both the non-threaded and threaded version of the API. If the solver is extended in the future to support role explanations and abducibles, also this part of the API implementation is already prepared.

5. Conclusions and Future Work

Based on our analysis of several ABox abduction solvers, we have proposed *DL Abduction API*, a Java API that may be implemented by DL abduction solvers in order to facilitate their integration into applications.

Our API allows to pass the most relevant inputs, including the background ontology, observations, and abducibles, and it features settings to toggle the most common options. Any additional specific options can be also passed in a format required specific to a given solver.

The inputs and outputs such as ontologies, axioms, or symbols are passed using OWL API [23] constructs as much as possible to facilitate the implementation for developers who are likely already acquainted with OWL API due to its popularity in the DL community.

As abduction is computationally demanding and explanations are found by exploring the given search space, our API includes a mechanism of incremental reporting of the found explanations that is based on the monitor design pattern.

The API is available as source code and as a `jar` file[2]. It has been already integrated into the latest version[3] of the experimental reasoner MHS-MXP [24]. Its integration into the LETHE reasoner [10, 4] is currently ongoing.

Acknowledgments

This work was supported by the Slovak Research and Development Agency under the Contract no. APVV-19-0220 (ORBIS) and by the EU H2020 programme under Contract no. 952215 (TAILOR). Martin Homola is also supported by projects VEGA 1/0621/22 and APVV-20-0353.

References

[1] C. S. Peirce, Illustrations of the logic of science VI: Deduction, induction, and hypothesis, Popular Science Monthly 13 (1878) 470–482.

[2]https://github.com/elratondesusi/DT
[3]https://github.com/elratondesusi/DT-demo

[2] C. Elsenbroich, O. Kutz, U. Sattler, A case for abductive reasoning over ontologies, in: Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, GA, US, volume 216 of *CEUR-WS*, 2006.

[3] J. Pukancová, M. Homola, Tableau-based ABox abduction for description logics: Preliminary report, in: Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016, 2016.

[4] P. Koopmann, Signature-based abduction with fresh individuals and complex concepts for description logics, in: Z. Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021, ijcai.org, 2021, pp. 1929–1935.

[5] S. Klarman, U. Endriss, S. Schlobach, ABox abduction in the description logic $\mathscr{ALC}$, Journal of Automated Reasoning 46 (2011) 43–80.

[6] K. Halland, K. Britz, Abox abduction in $\mathscr{ALC}$ using a DL tableau, in: 2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, 2012, pp. 51–58.

[7] F. Haifani, P. Koopmann, S. Tourret, C. Weidenbach, Connection-minimal abduction in $\mathscr{EL}$ via translation to FOL, in: Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2022), Lecture Notes on Computer Science, Springer, 2022. To appear.

[8] J. Du, H. Wan, H. Ma, Practical TBox abduction based on justification patterns, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 2017, pp. 1100–1106.

[9] F. Wei-Kleiner, Z. Dragisic, P. Lambrix, Abduction framework for repairing incomplete $\mathscr{EL}$ ontologies: Complexity results and algorithms, in: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada., 2014, pp. 1120–1127.

[10] P. Koopmann, W. Del-Pinto, S. Tourret, R. A. Schmidt, Signature-based abduction for expressive description logics, in: Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, 2020, pp. 592–602.

[11] G. Petasis, R. Möller, V. Karkaletsis, BOEMIE: Reasoning-based information extraction, in: Proceedings of the 1st Workshop on Natural Language Processing and Automated Reasoning co-located with 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013), A Corunna, Spain, September 15th, 2013., 2013, pp. 60–75.

[12] J. Du, G. Qi, Y. Shen, J. Z. Pan, Towards practical ABox abduction in large description logic ontologies, Int. J. Semantic Web Inf. Syst. 8 (2012) 1–33.

[13] İ. İ. Ceylan, T. Lukasiewicz, E. Malizia, C. Molinaro, A. Vaicenavicius, Explanations for negative query answers under existential rules, in: D. Calvanese, E. Erdem, M. Thielscher (Eds.), Proceedings of KR 2020, AAAI Press, 2020, pp. 223–232.

[14] D. Calvanese, M. Ortiz, M. Simkus, G. Stefanoni, Reasoning about explanations for negative query answers in DL-Lite, J. Artif. Intell. Res. 48 (2013) 635–669.

[15] K. Schekotihin, P. Rodler, W. Schmid, Ontodebug: Interactive ontology debugging plug-in for protégé, in: Foundations of Information and Knowledge Systems - 10th International

Symposium, FoIKS 2018, Budapest, Hungary, May 14-18, 2018, Proceedings, volume 10833 of *LNCS*, Springer, 2018, pp. 340–359.

[16] T. Hubauer, C. Legat, C. Seitz, Empowering adaptive manufacturing with interactive diagnostics: A multi-agent approach, in: Advances on Practical Applications of Agents and Multiagent Systems – 9th International Conference on Practical Applications of Agents and Multiagent Systems, PAAMS 2011, Salamanca, Spain, 2011, pp. 47–56.

[17] J. Pukancová, M. Homola, Abductive reasoning with description logics: Use case in medical diagnosis, in: Proceedings of the 28th International Workshop on Description Logics (DL 2015), Athens, Greece, volume 1350 of *CEUR-WS*, 2015.

[18] P. Koopmann, Two ways of explaining negative entailments in description logics using abduction, in: XLoKR 21, 2021.

[19] W. Del-Pinto, R. A. Schmidt, Forgetting-based abduction in $\mathcal{ALC}$, in: Proceedings of the Workshop on Second-Order Quantifier Elimination and Related Topics (SOQE 2017), Dresden, Germany, volume 2013 of *CEUR-WS*, 2017, pp. 27–35.

[20] W. Del-Pinto, R. A. Schmidt, Abox abduction via forgetting in ALC, in: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, AAAI Press, 2019, pp. 2768–2775.

[21] J. Pukancová, M. Homola, The AAA abox abduction solver, Künstliche Intell. 34 (2020) 517–522.

[22] M. Homola, J. Pukancová, J. Gablíková, K. Fabianová, Merge, explain, iterate, in: S. Borgwardt, T. Meyer (Eds.), Proceedings of the 33rd International Workshop on Description Logics (DL 2020), Online Event [Rhodes, Greece], volume 2663 of *CEUR-WS*, 2020.

[23] M. Horridge, S. Bechhofer, The OWL API: A Java API for OWL ontologies, Semantic Web 2 (2011) 11–21.

[24] M. Homola, J. Pukancová, I. Balintová, J. Boborová, Hybrid MHS-MXP ABox abduction solver: First empirical results, in: Proceedings od the 35rd International Workshop on Description Logics (DL 2022), Haifa, Israel, 2022.